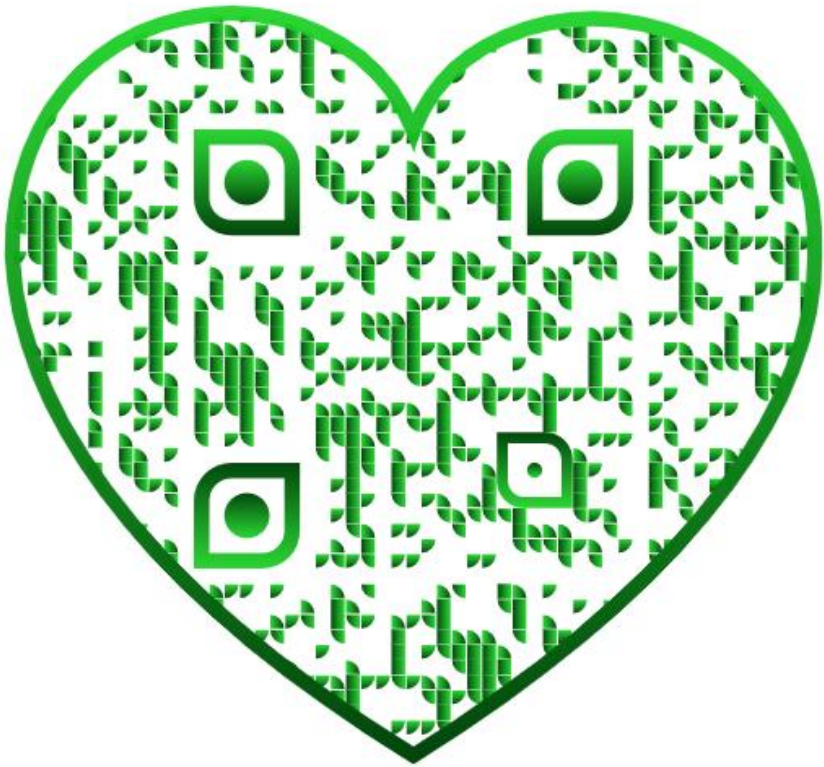
# Elixir IN ACTION

Saša Jurić

MANNING



# Elixir IN ACTION

Saša Jurić

MANNING

# Elixir in Action

**Saša Jurić**

# Copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: *orders@manning.com*

©2015 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Andrew Gibson
Copyeditor: Tiffany Taylor
Proofreader: Katie Tennant
Technical proofreader: Riza Fahmi
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781617292019

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 20 19 18 17 16 15

## Dedication

*To Renata, my wonderful wife*

# Brief Table of Contents


Copyright

Brief Table of Contents

Table of Contents

Preface

Acknowledgments

About this Book

About the Cover Illustration

1. The language

    Chapter 1. First steps

    Chapter 2. Building blocks

    Chapter 3. Control flow

    Chapter 4. Data abstractions

2. The platform

    Chapter 5. Concurrency primitives

    Chapter 6. Generic server processes

    Chapter 7. Building a concurrent system

    Chapter 8. Fault-tolerance basics

    Chapter 9. Isolating error effects

    Chapter 10. Sharing state

3. Production

    Chapter 11. Working with components

    Chapter 12. Building a distributed system

    Chapter 13. Running the system

Index

List of Figures

List of Tables

List of Listings

# Table of Contents


Copyright

Brief Table of Contents

Table of Contents

Preface

Acknowledgments

About this Book

About the Cover Illustration

1. The language

Chapter 1. First steps

1.1. About Erlang

1.1.1. High availability

1.1.2. Erlang concurrency

1.1.3. Server-side systems

1.1.4. The development platform

1.2. About Elixir

1.2.1. Code simplification

1.2.2. Composing functions

1.2.3. The big picture

1.3. Disadvantages

1.3.1. Speed

1.3.2. Ecosystem

1.4. Summary

Chapter 2. Building blocks

2.1. The interactive shell

2.2. Working with variables

2.3. Organizing your code

2.3.1. Modules

2.3.2. Functions

2.3.3. Function arity

2.3.4. Function visibility

2.3.5. Imports and aliases

2.3.6. Module attributes

2.3.7. Comments

2.4. Understanding the type system

2.4.1. Numbers

2.4.2. Atoms

2.4.3. Tuples

2.4.4. Lists

2.4.5. Immutability

2.4.6. Maps

2.4.7. Binaries and bitstrings

2.4.8. Strings

2.4.9. First-class functions

2.4.10. Other built-in types

2.4.11. Higher-level types

2.4.12. IO lists

2.5. Operators

2.6. Macros

2.7. Understanding the runtime

2.7.1. Modules and functions in the runtime

2.7.2. Starting the runtime

2.8. Summary

Chapter 3. Control flow

3.1. Pattern matching

3.1.1. The match operator

3.1.2. Matching tuples

3.1.3. Matching constants

3.1.4. Variables in patterns

3.1.5. Matching lists

3.1.6. Matching maps

3.1.7. Matching bitstrings and binaries

3.1.8. Compound matches

3.1.9. General behavior

3.2. Matching with functions

3.2.1. Multiclause functions

3.2.2. Guards

3.2.3. Multiclause lambdas

3.3. Conditionals

3.3.1. Branching with multiclause functions

3.3.2. Classical branching constructs

3.4. Loops and iterations

3.4.1. Iterating with recursion

3.4.2. Tail function calls

3.4.3. Higher-order functions

3.4.4. Comprehensions

3.4.5. Streams

3.5. Summary

Chapter 4. Data abstractions

4.1. Abstracting with modules

4.1.1. Basic abstraction

4.1.2. Composing abstractions

4.1.3. Structuring data with maps

4.1.4. Abstracting with structs

4.1.5. Data transparency

4.2. Working with hierarchical data

4.2.1. Generating IDs

4.2.2. Updating entries

4.2.3. Immutable hierarchical updates

4.2.4. Iterative updates

4.2.5. Exercise: importing from a file

4.3. Polymorphism with protocols

4.3.1. Protocol basics

4.3.2. Implementing a protocol

4.3.3. Built-in protocols

4.4. Summary

2. The platform

Chapter 5. Concurrency primitives

5.1. Principles

5.2. Working with processes

5.2.1. Creating processes
5.2.2. Message passing
5.3. Stateful server processes
5.3.1. Server processes
5.3.2. Keeping a process state
5.3.3. Mutable state
5.3.4. Complex states
5.3.5. Registered processes
5.4. Runtime considerations
5.4.1. A process is a bottleneck
5.4.2. Unlimited process mailboxes
5.4.3. Shared nothing concurrency
5.4.4. Scheduler inner workings
5.5. Summary
Chapter 6. Generic server processes
6.1. Building a generic server process
6.1.1. Plugging in with modules
6.1.2. Implementing the generic code
6.1.3. Using the generic abstraction
6.1.4. Supporting asynchronous requests
6.1.5. Exercise: refactoring the to-do server
6.2. Using gen_server
6.2.1. OTP behaviours
6.2.2. Plugging into gen_server
6.2.3. Handling requests

6.2.4. Handling plain messages

6.2.5. Other gen_server features

6.2.6. Process life cycle

6.2.7. OTP-compliant processes

6.2.8. Exercise: gen_server-powered to-do server

6.3. Summary

Chapter 7. Building a concurrent system

7.1. Working with the mix project

7.2. Managing multiple to-do lists

7.2.1. Implementing a cache

7.2.2. Analyzing process dependencies

7.3. Persisting data

7.3.1. Encoding and persisting

7.3.2. Using the database

7.3.3. Analyzing the system

7.3.4. Addressing the process bottleneck

7.3.5. Exercise: pooling and synchronizing

7.4. Reasoning with processes

7.5. Summary

Chapter 8. Fault-tolerance basics

8.1. Runtime errors

8.1.1. Error types

8.1.2. Handling errors

8.2. Errors in concurrent systems

8.2.1. Linking processes

8.2.2. Trapping exits

8.2.3. Monitors

8.3. Supervisors

8.3.1. Supervisor behaviour

8.3.2. Defining a supervisor

8.3.3. Starting a supervisor

8.3.4. Linking all processes

8.3.5. Restart frequency

8.4. Summary

Chapter 9. Isolating error effects

9.1. Supervision trees

9.1.1. Separating loosely dependent parts

9.1.2. Rich process discovery

9.1.3. Supervising database workers

9.1.4. Removing the database process

9.1.5. Starting the system

9.1.6. Organizing the supervision tree

9.2. Starting workers dynamically

9.2.1. simple_one_for_one supervisors

9.2.2. Restart strategies

9.3. “Let it crash”

9.3.1. Error kernel

9.3.2. Handling expected errors

9.3.3. Preserving the state

9.4. Summary

Chapter 10. Sharing state

10.1. Single-process bottleneck

10.2. ETS tables

10.2.1. Basic operations

10.2.2. ETS powered page cache

10.2.3. Other ETS operations

10.2.4. Other use cases for ETS

10.2.5. Beyond ETS

10.2.6. Exercise: ETS-powered process registry

10.3. Summary

3. Production

Chapter 11. Working with components

11.1. OTP applications

11.1.1. Describing an application

11.1.2. Creating applications with the mix tool

11.1.3. The application behaviour

11.1.4. Starting the application

11.1.5. Library applications

11.1.6. Creating a to-do application

11.1.7. The application folder structure

11.2. Working with dependencies

11.2.1. Specifying dependencies

11.2.2. Adapting the registry

11.3. Building a web server

11.3.1. Choosing dependencies

11.3.2. Starting the server

11.3.3. Handling requests

11.3.4. Reasoning about the system

11.4. Managing the application environment

11.5. Summary

Chapter 12. Building a distributed system

12.1. Distribution primitives

12.1.1. Starting a cluster

12.1.2. Communicating between nodes

12.1.3. Process discovery

12.1.4. Links and monitors

12.1.5. Other distribution services

12.2. Building a fault-tolerant cluster

12.2.1. Cluster design

12.2.2. The distributed to-do cache

12.2.3. Implementing a replicated database

12.2.4. Testing the system

12.2.5. Dealing with partitions

12.2.6. Highly available systems

12.3. Network considerations

12.3.1. Node names

12.3.2. Cookies

12.3.3. Hidden nodes

12.3.4. Firewalls

12.4. Summary

Chapter 13. Running the system

13.1. Running a system with the mix tool

13.1.1. Bypassing the shell

13.1.2. Running as a background process

13.1.3. Running scripts

13.1.4. Compiling for production

13.2. OTP releases

13.2.1. Building a release with exrm

13.2.2. Using a release

13.2.3. Release contents

13.3. Analyzing system behavior

13.3.1. Debugging

13.3.2. Logging

13.3.3. Interacting with the system

13.3.4. Tracing

13.4. Summary

Index

List of Figures

List of Tables

List of Listings

# Preface

In 2010, I was given the task of implementing a system to transmit frequent updates to a few thousand connected users in near real time. My company at the time was mostly using Ruby on Rails. But for such a highly concurrent challenge, I needed something more suitable. Following the suggestion of my CTO, I looked into Erlang, read some material, made a prototype, and performed a load test. I was impressed with the initial results and moved on to implement the whole thing in Erlang. A few months later, the system was shipped, and it's been doing its job ever since.

As time passed, I began to increasingly appreciate Erlang and the way it helped me manage such a complex system. Gradually, I came to prefer Erlang over the technologies I had used previously. I began to evangelize the language, first in my company and then at local events, then, finally, at the end of 2012, I started the blog The Erlangelist (http://theerlangelist.com), where I aim to showcase the advantages of Erlang to programmers from OO backgrounds.

Because Erlang is an unusual language, I began experimenting with Elixir, hoping it would help me explain the beauty of Erlang in a way that would resonate with OO programmers. Despite the fact that it was at an early stage of development (at the time, it was at version 0.8), I was immediately impressed with Elixir's maturity and the way it integrated with Erlang. Soon, I started using Elixir to develop new features for my Erlang-based system.

A few months later, I was contacted by Michael Stephens of Manning, who asked me if I was interested in writing a book about Elixir. At the time, two Elixir books were already in the making. After some consideration, I decided there was space for another book that would approach the topic from a different angle. In the proposal for this book, I wrote, "Essentially, I am proposing a book which I would have personally found useful when I started learning Erlang a couple of years ago." In other words, this book aims to bring programmers new to Elixir and Erlang to the point where they can develop complex systems on their own. In the process, I tried to recollect the mistakes I had made in Erlang, most of which were due to my OO background

and inexperience with the language. This book tries to point you in the right direction so you don't end up making those same mistakes.

Almost two years later, I'm excited the work is finally done. I hope you'll enjoy reading the book, learn a lot from it, and have a chance to apply your new knowledge in your work!

# Acknowledgments

Writing this book required a significant investment of my time, so, above all, I want to thank my wife Renata for her endless support and patience during those long hours and busy weekends.

I'd like to thank Manning for giving me the chance to write this book. In particular, thanks to Michael Stephens for making the initial contact, to Marjan Bace for giving me a chance to write this book, to Bert Bates for setting me in the right direction, to Karen Miller for keeping me on track, to Aleksandar Dragosavljevic for managing the review process, to Kevin Sullivan for overseeing production, to Tiffany Taylor for transforming my "English" into proper English, and to Candace Gillhoolley and Ana Romac for promoting the book.

The material in this book has been significantly improved thanks to great feedback from reviewers and early readers. Above all, I wish to thank Andrew Gibson for useful feedback, great insights, and for rescuing me when I got stuck at the last hurdle. I'd also like to thank Alexei Sholik and Peter Minten, who provided excellent immediate technical feedback during the writing process.

A big thank you to all the technical reviewers: Al Rahimi, Alan Lenton, Alexey Galliulin, Christopher Bailey, Christopher Haupt, Heather Campbell, Jeroen Benckhuijsen, Kosmas Chatzimichalis, Mark Ryall, Mohsen Mostafa Jokar, Tom Geudens, Tomer Elmalem, and Ved Antani. Special thanks to Riza Fahmi who reviewed the manuscript carefully one last time shortly before it went into production.

I also wish to thank all the readers who bought and read the MEAP (Manning Early Access Program) version and provided useful comments. Thank you for taking the time to read my ramblings and for providing insightful feedback.

The people who gave us Elixir and Erlang, including the original inventors, core team members, and contributors, deserve a special mention. Thank you for creating such great products, which make my

job easier and more fun. Finally, special thanks to all the members of the Elixir community; this is the nicest and friendliest developer community I've ever seen!

# About this Book

Elixir is a modern functional programming language for building large-scale scalable, distributed, fault-tolerant systems for the Erlang virtual machine. Although the language is compelling in its own right, arguably its biggest advantage is that it targets the Erlang platform.

Erlang was made to help developers deal with the challenge of high availability. Originally, the product was intended for development of telecommunication systems; but today it's used in all kinds of domains, such as collaboration tools, online payment systems, real-time bidding systems, database servers, and multiplayer online games, to name only a few examples. If you're developing a system that must provide service to a multitude of users around the world, you want that system to function continuously, without noticeable downtime, regardless of any software or hardware problems that occur at runtime. Otherwise, significant and frequent outages will leave end users unhappy, and ultimately they may seek alternative solutions. A system with frequent downtime is unreliable and unusable, and thus fails to fulfill its intended purpose. Therefore, high availability becomes an increasingly important property—and Erlang can help you achieve that.

Elixir aims to modernize and improve the experience of developing Erlang-powered systems. The language is a compilation of features from various other languages such as Erlang, Clojure, and Ruby. Furthermore, Elixir ships with a toolset that simplifies project management, testing, packaging, and documentation building. Arguably, Elixir lowers the entry barrier into the Erlang world and improves developer productivity. Having the Erlang runtime as the target platform means Elixir-based systems are able to use all the libraries from the Erlang ecosystem, including the battle-tested OTP framework that ships with Erlang.

This book is a tutorial that will teach you how to build production-ready, Elixir-based systems. It's not a complete reference on Elixir and Erlang—it doesn't cover every nuance of the language or every possible aspect of the underlying Erlang VM. It glosses over or skips many topics, such as floating-point precision, Unicode specifics, file I/

O, unit testing, and more. Although they're relevant, such topics aren't the book's primary focus, and you can research them yourself when the need arises. Omitting or quickly dealing with these conventional topics gives us space to treat more interesting and unusual areas in greater detail. Concurrent programming and the way it helps bring scalability, fault tolerance, distribution, and availability to systems—these are the core topics of this book.

Even the techniques that are covered aren't treated in 100% detail. I've omitted some fine-print nuances for the sake of brevity and focus. The goal is not to provide complete coverage, but rather to teach you about the underlying principles and how each piece fits into the bigger picture. After finishing the book, it should be simple to research and understand the remaining details on your own. To give you a push in the right direction, mentions and links to further interesting topics appear throughout the book.

Because the book deals with upper-intermediate topics, there are some prerequisites you should meet before reading it. I assume you're a professional software developer with a couple years of experience. The exact technology you're proficient in isn't relevant: it can be Java, C#, Ruby, C++, or another general-purpose programming language. Any experience in development of backend (server-side) systems is welcome.

You don't need to know anything about Erlang, Elixir, or other concurrent platforms. In particular, you don't need to know anything about functional programming. Elixir is a functional language, and if you come from an OO background, this may scare you a bit. As a long-time OO programmer, I can sincerely tell you not to worry. The underlying functional concepts in Elixir are relatively simple and should be easy to grasp. Of course, functional programming is significantly different from whatever you've seen in a typical OO language, and it takes some time getting used to. But it's not rocket science, and if you're an experienced developer, you should have no problem understanding these concepts.

## Roadmap

The book is divided into three parts.

Part 1 introduces the Elixir language, presents and describes its basic building blocks, and then treats common functional programming idioms in more detail:

- Chapter 1 provides a high-level overview of Erlang and Elixir and explains why those technologies are useful and what distinguishes them from other languages and platforms.
- Chapter 2 presents the main building blocks of Elixir, such as modules, functions, and the type system.
- Chapter 3 gives a detailed explanation of pattern matching and how it's used to deal with flow control.
- Chapter 4 explains how to build higher-level abstractions on top of immutable data structures.

Part 2 builds on these foundations and focuses on the primary aspects of the Erlang virtual machine. You'll learn about the Erlang concurrency model and its many benefits, such as scalability and fault-tolerance:

- Chapter 5 explains the Erlang concurrency model and presents basic concurrency primitives.
- Chapter 6 discusses generic server processes: building blocks of highly concurrent Elixir/Erlang systems.
- Chapter 7 demonstrates how to build a more involved concurrent system.
- Chapter 8 presents the idioms of error handling, with a special focus on errors and faults in concurrent systems.
- Chapter 9 provides an in-depth discussion of how to isolate all kinds of errors and minimize their impact in production.
- Chapter 10 discusses ETS tables and explains how they can be used to maintain a system-wide state in a highly concurrent system.

Part 3 deals with systems in production:

- Chapter 11 explains OTP applications, which are used to package reusable components.
- Chapter 12 discusses distributed systems, which can help you

improve fault tolerance and scalability.
- Chapter 13 presents various ways of preparing an Elixir-based system for production, focusing in particular on OTP releases.

## Code conventions and downloads

All source code in this book is in a `fixed-width font like this`, which sets it off from the surrounding text. In many listings, the code is annotated to point out key concepts. We have tried to format the code so that it fits within the available page space in the book by adding line breaks and using indentation carefully.

The code accompanying this book is hosted at the GitHub repository: https://github.com/sasa1977/elixir-in-action. It is also available for download as a zip file from the publisher's website at www.manning.com/ElixirinAction.

## Author Online

Purchase of *Elixir in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. To access the forum and subscribe to it, point your web browser to www.manning.com/ElixirinAction. This Author Online (AO) page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog among individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The AO forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

## About the author

Saša Jurić is a developer with extensive experience implementing high-volume, concurrent, server-side systems; Windows desktop applications; and kiosk applications. After almost 20 years of object-oriented programming, he discovered Erlang and Elixir. He used both languages to build a scalable, fault-tolerant HTTP push server and the supporting backend system. Currently he's part of the Aircloak team and is using Erlang to build a plug-and-play privacy-compliance solution. He occasionally blogs about Elixir and Erlang at http://theerlangelist.com.

# About the Cover Illustration

The figure on the cover of *Elixir in Action* is captioned "A Russian Girl." The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771), was called "Geographer to King George III." An English cartographer, he was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. Jeffreys' work as a map maker sparked an interest in local dress customs of the lands he surveyed and brilliantly displayed in this four-volume collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late eighteenth century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jeffreys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys' illustrations.

# Part 1. The language

The first part of the book is an introduction to the Elixir language. We start by providing a high-level overview of Elixir and Erlang, discussing the goals and benefits of both technologies. In chapter 2, you'll learn about the basic building blocks of the Elixir language, such as modules, functions, and the type system. Chapter 3 details the treatment of pattern-matching and control-flow idioms. In chapter 4, you'll learn how to implement higher-level data abstractions with immutable data structures.

# Chapter 1. First steps

*This chapter covers*

- Overview of Erlang
- Benefits of Elixir

This is the beginning of your journey into the world of Elixir and Erlang, two efficient and useful technologies that can significantly simplify the development of large, scalable systems. Chances are, you're reading this book to learn about Elixir. But because Elixir is built on top of Erlang and depends heavily on it, you should first learn a bit about what Erlang is and the benefits it offers. So, let's take a brief, high-level look at Erlang.

## 1.1. About Erlang

Erlang is a development platform for building scalable and reliable systems that constantly provide service with little or no downtime. This is a bold statement, but it's exactly what Erlang was made for. Conceived in the mid-1980s by Ericsson, a Swedish telecom giant, Erlang was driven by the needs of the company's own telecom systems, where properties like reliability, responsiveness, scalability, and constant availability are imperative. A telephone network should always operate regardless of the number of simultaneous calls, unexpected bugs, or hardware and software upgrades taking place.

Despite being originally built for telecom systems, Erlang is in no way specialized for this domain. It doesn't contain explicit support for programming telephones, switches, or other telecom devices. Instead, it's a general-purpose development platform that provides special support for technical, nonfunctional challenges such as concurrency, scalability, fault-tolerance, distribution, and high availability.

In late 1980s and early 90s, when most software was desktop based, the need for high availability was limited to specialized systems such

as telecoms. Today we face a much different situation: the focus is on the internet and the Web, and most applications are driven and supported by a server system that processes requests, crunches data, and pushes relevant informations to many connected clients. Today's popular systems are more about communication and collaboration; examples include social networks, content-management systems, on-demand multimedia, and multiplayer games.

All of these systems have some common nonfunctional requirements. The system must be responsive, regardless of the number of connected clients. The impact of unexpected errors must be as minimal as possible, instead of affecting the entire system. It's acceptable if an occasional request fails due to a bug, but it's a major problem when the entire system becomes completely unavailable. Ideally, the system should never crash or be taken down, not even during a software upgrade. It should always be up and running, providing the service to its clients.

These goals might seem difficult, but they're imperative when building systems that people depend on. Unless a system is responsive and reliable, it will eventually fail to fulfill its purpose. Therefore, when building server-side systems, it's essential to make the system constantly available.

And this is the intended purpose of Erlang. High availability is explicitly supported via technical concepts such as scalability, fault tolerance, and distribution. Unlike with most other modern development platforms, these concepts were the main motivation and driving force behind the development of Erlang. The Ericsson team, led by Joe Armstrong, spent a couple of years designing, prototyping, and experimenting before creating the development platform today known as Erlang. Its uses may have been limited in early 90s, but today almost any system can benefit from it.

Erlang has recently gained more attention. It powers various large systems and has been doing so for more than two decades, such as the WhatsApp messaging application, the Riak distributed database, the Heroku cloud, the Chef deployment automation system, the RabbitMQ message queue, financial systems, and multiplayer backends. It's truly a proven technology, both in time and scale. But what is the magic behind Erlang? Let's take a look at how Erlang can help you build

highly available, reliable systems.

### 1.1.1. High availability

Erlang was specifically created to support the development of highly available systems—systems that are always online and provide service to their clients even when faced with unexpected circumstances. On the surface, this may seem simple, but as you probably know, many things can go wrong in production. To make systems work 24/7 without any downtime, we have to tackle some technical challenges:

- ***Fault tolerance—*** A system has to keep working when something unforeseen happens. Unexpected errors occur, bugs creep in, components occasionally fail, network connections drop, or the entire machine where the system is running crashes. Whatever happens, we want to localize the impact of an error as much as possible, recover from the error, and keep the system running and providing service.
- ***Scalability—*** A system should be able to handle any possible load. Of course, we don't buy tons of hardware just in case the entire planet's population might start using our system some day. But we should be able to respond to a load increase by adding more hardware resources without any software intervention. Ideally, this should be possible without a system restart.
- ***Distribution—*** To make a system that never stops, we have to run it on multiple machines. This promotes the overall stability of the system: if a machine is taken down, another one can take over. Furthermore, this gives us means to scale horizontally—we can address load increase by adding more machines to the system, thus adding work units to support the higher demand.
- ***Responsiveness—*** It goes without saying that a system should always be reasonably fast and responsive. Request handling shouldn't be drastically prolonged, even if the load increases or unexpected errors happen. In particular, occasional lengthy tasks shouldn't block the rest of the system or have a significant effect on performance.
- ***Live update—*** In some cases, you may want to push a new version of your software without restarting any servers. For example, in a telephony system, we don't want to disconnect established calls while we upgrade the software.

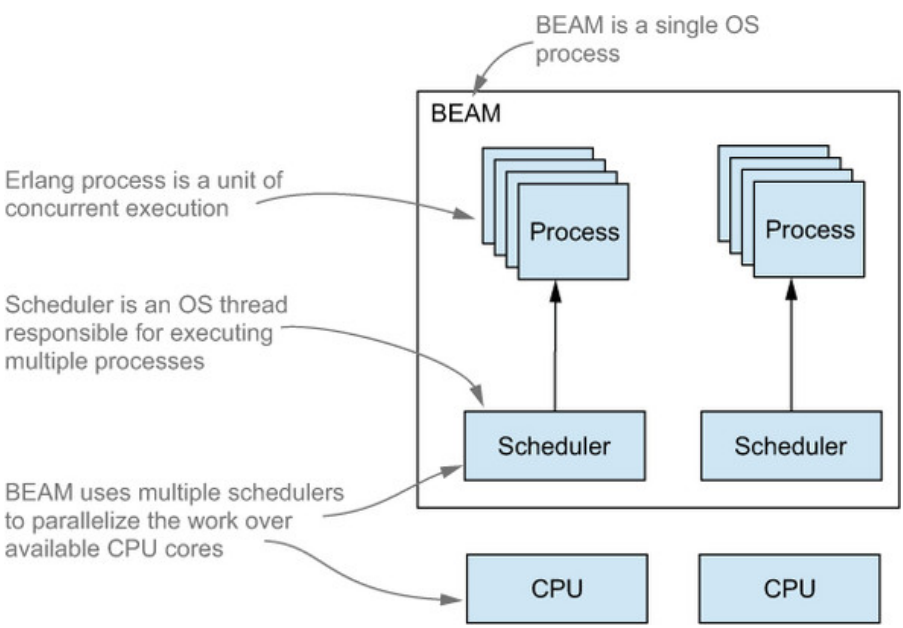
If we manage to handle these challenges, the system will truly become highly available and be able to constantly provide service to users, rain or shine.

Erlang gives us tools to address these challenges—that is what it was built for. A system can gain all these properties and ultimately become highly available through the power of the Erlang concurrency model. Next, let's look at how concurrency works in Erlang.

### 1.1.2. Erlang concurrency

Concurrency is at the heart and soul of Erlang systems. Almost every nontrivial Erlang-based production system is highly concurrent. Even the programming language is sometimes called a *concurrency-oriented language*. Instead of relying on heavyweight threads and OS processes, Erlang takes concurrency into its own hands, as illustrated in figure 1.1.

**Figure 1.1. Concurrency in the Erlang virtual machine**




The basic concurrency primitive is called an *Erlang process* (not to be confused with OS processes or threads), and typical Erlang systems

run thousands or even millions of such processes. The Erlang virtual machine, called BEAM[1], uses its own schedulers to distribute the execution of processes over the available CPU cores, thus parallelizing execution as much as possible. The way processes are implemented provides many benefits.

1 Bogdan/Björn's Erlang Abstract Machine.

**Fault tolerance**

Erlang processes are completely isolated from each other. They share no memory, and a crash of one process doesn't cause a crash of other processes. This helps you isolate the effect of an unexpected error. If something bad happens, it has only a local impact. Moreover, Erlang provides you with means to detect a process crash and do something about it: typically, you start a new process in place of the crashed one.

**Scalability**

Sharing no memory, processes communicate via asynchronous messages. This means there are no complex synchronization mechanisms such as locks, mutexes, or semaphores. Consequently, the interaction between concurrent entities is much simpler to develop and understand. Typical Erlang systems are divided into a large number of concurrent processes, which cooperate together to provide the complete service. The virtual machine can efficiently parallelize the execution of processes as much as possible. This makes Erlang systems scalable, because they can take advantage of all available CPU cores.

**Distribution**

Communication between processes works the same way regardless of whether these processes reside in the same BEAM instance or on two different instances on two separate, remote computers. Therefore, a typical highly concurrent Erlang-based system is automatically ready to be distributed over multiple machines. This in turn gives you the ability to scale out—to run a cluster of machines that share the total system load. Additionally, running on multiple machines makes the system truly resilient—if one machine crashes, others can take over.

**Responsiveness**

Runtime is specifically tuned to promote overall responsiveness of the system. I've mentioned that Erlang takes the execution of multiple processes into its own hands by employing dedicated schedulers that interchangeably execute many Erlang processes. A scheduler is preemptive—it gives a small execution window to each process and then pauses it and runs another process. Because the execution window is small, a single long-running process can't block the rest of the system. Furthermore, I/O operations are internally delegated to separate threads, or a kernel-poll service of the underlying OS is used if available. This means any process that waits for an I/O operation to finish won't block the execution of other processes.

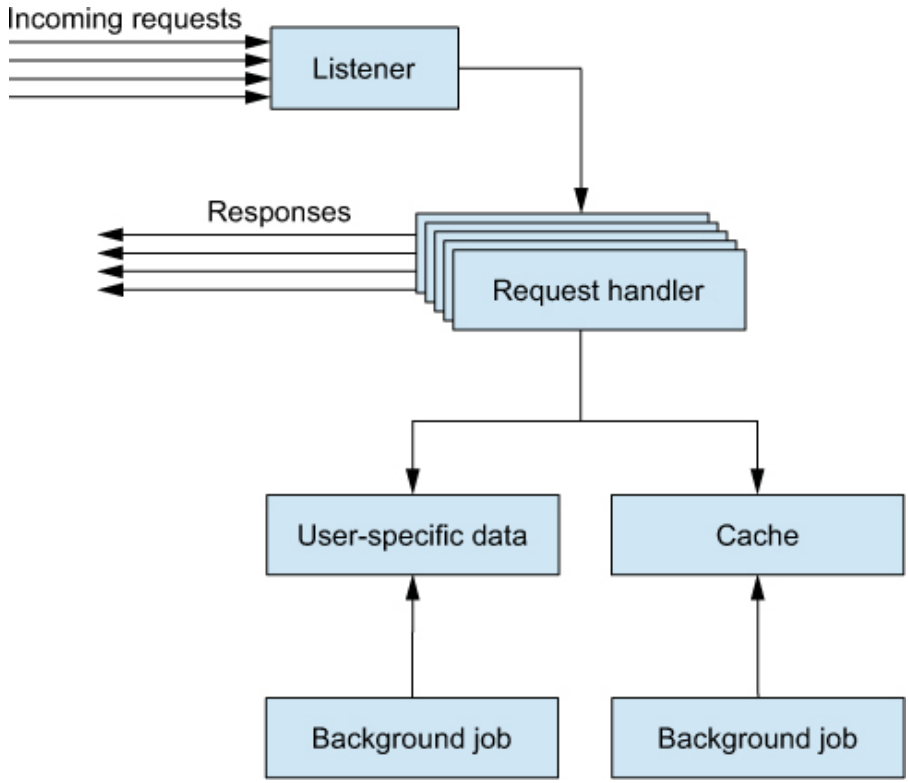
Even garbage collection is specifically tuned to promote system responsiveness. Recall that processes are completely isolated and share no memory. This allows per-process garbage collection: instead of stopping the entire system, each process is individually collected as needed. Such collections are much shorter and don't block the entire system for long periods of time. In fact, in a multicore system, it's possible for one CPU core to run a short garbage collection while the remaining cores are doing standard processing.

As you can see, concurrency is a crucial element in Erlang; and it's related to more than just parallelism. Owing to the underlying implementation, concurrency promotes fault tolerance, distribution, and system responsiveness. Typical Erlang systems run many concurrent tasks, using thousands or even millions of processes. This can be especially useful when you're developing server-side systems, which can often be implemented completely in Erlang.

### 1.1.3. Server-side systems

Erlang can be used in various applications and systems. There are examples of Erlang-based desktop applications, and it's often used in embedded environments. Its sweet spot, in my opinion, lies in server-side systems—systems that run on one or more server and must serve many simultaneous clients. The term *server-side system* indicates that it's more than a simple server that processes requests. It's an entire system that, in addition to request handling, must run various background jobs and manage some kind of server-wide in-memory state, as illustrated in figure 1.2.

**Figure 1.2. Server-side system**




A server-side system is often distributed on multiple machines that collaborate to produce business value. You might place different components on different machines, and you also might deploy some components on multiple servers to achieve load balancing and/or support failover scenarios.

This is where Erlang can make your life significantly simpler. By giving you primitives to make your code concurrent, scalable, and distributed, it allows you to implement the entire system completely in Erlang. Every component in figure 1.2 can be implemented as an Erlang process! This makes the system scalable, fault tolerant, and easy to distribute. By relying on Erlang's error-detection and recovery primitives, you can further increase reliability and recover from unexpected errors.

Let's look at a real-life example. I have been involved professionally in the development of two web servers, both of which have similar

technical needs: they serve a multitude of clients, handle long-running requests, manage server-wide in-memory state, persist data that must survive OS process and machine restarts, and run background jobs. Table 1.1 lists the technologies used in each server:

**Table 1.1. Comparison of technologies used in two real-life web servers**

| Technical requirement | Server A | Server B |
|---|---|---|
| HTTP server | Nginx and Phusion Passenger | Erlang |
| Request processing | Ruby on Rails | Erlang |
| Long-running requests | Go | Erlang |
| Server-wide state | Redis | Erlang |
| Persistable data | Redis and MongoDB | Erlang |
| Background jobs | Cron, Bash scripts, and Ruby | Erlang |
| Service crash recovery | Upstart | Erlang |

Server A is powered by various technologies, most of them known and popular in the community. There were specific reasons for using these technologies: each was introduced to resolve a shortcoming of those already present in the system. For example, Ruby on Rails handles concurrent requests in separate OS processes. We needed a way to share data between these different processes, so we introduced Redis. Similarly, MongoDB is used to manage persistent front-end data, most often user-related information. Thus there is a rationale behind every technology used in server A, but the entire solution seems complex. It's not contained in a single project, the components are deployed separately, and it isn't trivial to start the entire system on a development machine. We had to develop a tool to help us start the system locally!

In contrast, server B accomplishes the same technical requirements while relying on a single technology, using platform features created specifically for these purposes and proven in large systems. Moreover, the entire server is a single project that runs inside a single BEAM instance—in production, it runs inside only one OS process, using a handful of OS threads. Concurrency is handled completely by the Erlang scheduler, and the system is scalable, responsive, and fault tolerant. Because it's implemented as a single project, the system is easier to manage, deploy, and run locally on the development machine.

It's important to notice that Erlang tools aren't always full-blown alternatives to mainstream solutions, such as web servers like Nginx, database servers like Riak, and in-memory key-value stores like Redis. But Erlang gives you options, making it possible to implement an

initial solution using exclusively Erlang and resort to alternative technologies when an Erlang solution isn't sufficient. This makes the entire system more homogeneous and therefore easier to develop and maintain.

It's also worth noting that Erlang isn't an isolated island. It can run in-process C code and can communicate with practically any external component such as message queues, in-memory key-value stores, and external databases. Therefore, when opting for Erlang, you aren't deprived of using existing third-party technologies. Instead, you have the option of using them when they're called for, not because your primary development platform doesn't give you a tool to solve your problems.

Now that you know about Erlang's strengths and the areas where it excels, let's take a closer look at what Erlang *is*.

### 1.1.4. The development platform

Erlang is more than a programming language. It's a full-blown development platform consisting of four distinct parts: the language, the virtual machine, the framework, and the tools.

Erlang, the language, is the primary way of writing code that runs in the Erlang virtual machine. It's a simple, functional language with basic concurrency primitives.

Source code written in Erlang is compiled into byte code that is then executed in the BEAM. This is where the true magic happens. The virtual machine parallelizes your concurrent Erlang programs and takes care of process isolation, distribution, and overall responsiveness of the system.

The standard part of the release is a framework called *Open Telecom Platform (*OTP*)*. Despite its somewhat unfortunate name, the framework has nothing to do with telecom systems. It's a general-purpose framework that abstracts away many typical Erlang tasks:

- Concurrency and distribution patterns
- Error detection and recovery in concurrent systems
- Packaging code into libraries
- Systems deployment
- Live code updates

All these things can be done without OTP, but that makes no sense. OTP is battle tested in many production systems and is such an integral part of Erlang that it's hard to draw a line between the two. Even the official distribution is called Erlang/OTP.

The tools are used for various typical tasks such as compiling Erlang code, starting a BEAM instance, creating deployable releases, running the interactive shell, connecting to the running BEAM instance, and so on. Both BEAM and its accompanying tools are cross platform. You can run them on most mainstream operating systems, such as Unix, Linux, and Windows. The entire Erlang distribution is open source, and you can find the source on the official site (http://erlang.org) or on the Erlang GitHub repository (https://github.com/erlang/otp). Ericsson is still in charge of the development process and releases a new version on a regular basis, once a year.

That concludes the story of Erlang. But if Erlang is so great, why do you need Elixir? The next section aims to answer this question.

## 1.2. About Elixir

Elixir is an alternative language for the Erlang virtual machine that allows you to write cleaner, more compact, code that does a better job of revealing your intentions. You write programs in Elixir and run them normally in BEAM.

Elixir is an open source project, originally started by José Valim. Unlike Erlang, Elixir is more of a collaborative effort; presently it has about 200 contributors. New features are frequently discussed on mailing lists, the GitHub issue tracker, and the #elixir-lang freenode IRC channel. José has the last word, but the entire project is a true open source collaboration, attracting an interesting mixture of

seasoned Erlang veterans and talented young developers. The source code can be found on the GitHub repository at https://github.com/elixir-lang/elixir.

Elixir targets the Erlang runtime. The result of compiling the Elixir source code are BEAM-compliant byte-code files that can run in a BEAM instance and can normally cooperate with pure Erlang code—you can use Erlang libraries from Elixir and vice versa. There is nothing you can do in Erlang that can't be done in Elixir, and usually the Elixir code is as performant as its Erlang counterpart.

Elixir is semantically close to Erlang: many of its language constructs map directly to the Erlang counterparts. But Elixir provides some additional constructs that make it possible to radically reduce boilerplate and duplication. In addition, it tidies up some important parts of the standard libraries and provides some nice syntactic sugar and a uniform tool for creating and packaging systems. Everything you can do in Erlang is possible in Elixir, and vice versa; but in my experience, the Elixir solution is usually easier to develop and maintain.

Let's take a closer look at how Elixir improves on some Erlang features. We'll start with boilerplate and noise reduction.

### 1.2.1. Code simplification

One of the most important benefits of Elixir is the ability to radically reduce boilerplate and eliminate noise from code, which results in simpler code that is easier to write and maintain. Let's see what this means by contrasting Erlang and Elixir code. A frequently used building block in Erlang concurrent systems is the *server process*. You can think of server processes as something like concurrent objects—they embed private state and can interact with other processes via messages. Being concurrent, different processes may run in parallel. Typical Erlang systems rely heavily on processes, running thousands or even millions of them. The following example Erlang code implements a simple server process that adds two numbers.

**Listing 1.1. Erlang-based server process that adds two numbers**

```
-module(sum_server).

-behaviour(gen_server).
-export([
  start/0, sum/3,
  init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
  code_change/3
]).

start() -> gen_server:start(?MODULE, [], []).
sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).

init(_) -> {ok, undefined}.

handle_call({sum, A, B}, _From, State) -> {reply, A + B, State};
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Even without any knowledge of Erlang, this seems like a lot of code for something that only adds two numbers. To be fair, the addition is concurrent; but regardless, due to the amount of code, it's hard to see the forest for the trees. It's definitely not immediately obvious what the code does. Moreover, it's difficult to write such code. Even after years of production-level Erlang development, I still can't write this without consulting the documentation or copying and pasting it from previously written code.

The problem with Erlang is that this boilerplate is almost impossible to remove, even if it's identical in most places (which in my experience is the case). The language provides almost no support for eliminating this noise. In all fairness, there is a way to reduce the boilerplate using a construct called `parse transform`, but it's clumsy and complicated to use. So in practice, Erlang developers write their server processes using the just-presented pattern.

Because server processes are an important and frequently used tool in Erlang, it's an unfortunate fact that Erlang developers have to constantly copy-paste this noise and work with it. Surprisingly, many people get used to it, probably due to the wonderful things BEAM does for them. It's often said that Erlang makes hard things easy and easy things hard. Still, the previous code leaves an impression that you should be able to do better.

Let's see the Elixir version of the same server process, presented in the next listing.

**Listing 1.2. Elixir-based server process that adds two numbers**

```
defmodule SumServer do
  use GenServer

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def sum(server, a, b) do
    GenServer.call(server, {:sum, a, b})
  end

  def handle_call({:sum, a, b}, _from, state) do
    {:reply, a + b, state}
  end
end
```

This code is significantly smaller and therefore easier to read and maintain. Its intention is more clearly revealed, it's less burdened with noise. And yet it's as capable and flexible as the Erlang version. It behaves exactly the same at runtime and retains the complete semantics. There is nothing you can do in the Erlang version that's not possible in its Elixir counterpart.

Despite being significantly smaller, the Elixir version of a sum server process still feels somewhat noisy, given that all it does is add two numbers. The excess noise exists because Elixir retains a 1:1 semantic relation to the underlying Erlang library that is used to create server processes.

But the language gives you tools to further eliminate whatever you may regard as noise and duplication. For example, I have developed my own Elixir library called ExActor that makes the server process definition dense, as shown next.

**Listing 1.3. Elixir-based server process to add numbers based on a third-party abstraction**

```
defmodule SumServer do
  use ExActor.GenServer

  defstart start

  defcall sum(a, b) do
    reply(a + b)
```

```
  end
end
```

The intention of this code should be obvious even to developers with no previous Elixir experience. At runtime, the code works almost completely the same as the two previous versions. The transformation that makes this code behave like the previous ones happens at compile time. When it comes to the byte code, all three versions are similar.

**Note**

I mention the ExActor library only to illustrate how much you can abstract away in Elixir. But you won't use this library in this book, because it's a third-party abstraction that hides important details of how server processes work. To completely take advantage of server processes, it's important that you understand what makes them tick, which is why in this book you'll learn about lower-level abstractions. Once you understand how server processes work, you can decide for yourself whether you want to use ExActor to implement server processes.

This last implementation of the `sum` server process is powered by the Elixir macros facility. A *macro* is Elixir code that runs at *compile time*. A macro takes an internal representation of your source code as input and can create alternative output. Elixir macros are inspired by LISP and should not be confused with C-style macros. Unlike C/C++ macros, which work with pure text, Elixir macros work on abstract syntax tree (AST) structure, which makes it easier to perform nontrivial manipulations of the input code to obtain alternative output. Of course, Elixir provides helper constructs to simplify this transformation.

Take another look at how the `sum` operation is defined in the last example:

```
defcall sum(a, b) do
  reply(a + b)
end
```

Notice the `defcall` at the beginning. There is no such keyword in Elixir. This is a custom macro that translates the given definition to something like the following:

```
def sum(server, a, b) do
  GenServer.call(server, {:sum, a, b})
end

def handle_call({:sum, a, b}, _from, state) do
  {:reply, a + b, state}
end
```

Because macros are written in Elixir, they're flexible and powerful, making it possible to extend the language and introduce new constructs that look like an integral part of the language. For example, the open source Ecto project, which aims to bring LINQ style queries to Elixir, is also powered by Elixir macro support and provides an expressive query syntax that looks deceptively like the part of the language:

```
from w in Weather,
  where: w.prcp > 0 or w.prcp == nil,
  select: w
```

Due to its macro support and smart compiler architecture, most of Elixir is written in Elixir. Language constructs like `if` and `unless` and support for structures are implemented via Elixir macros. Only the smallest possible core is done in Erlang—everything else is then built on top of it in Elixir!

Elixir macros are something of a black art, but they make it possible to flush out nontrivial boilerplate at compile time and extend the language with your own DSL-like constructs. But Elixir isn't all about macros. Another worthy improvement is some seemingly simple syntactic sugar that makes functional programming much easier.

### 1.2.2. Composing functions

Both Erlang and Elixir are functional languages. They rely on immutable data and functions that transform data. One of the supposed benefits of this approach is that code is divided into many small, reusable, composable functions.

Unfortunately, the composability feature works clumsily in Erlang. Let's look at an adapted example from my own work. One piece of

code I'm responsible for maintains an in-memory model and receives XML messages that modify the model. When an XML message arrives, the following actions must be done:

1. Apply the XML to the in-memory model.
2. Process the resulting changes.
3. Persist the model.

Here's an Erlang sketch of the corresponding function:

```
process_xml(Model, Xml) ->
  Model1 = update(Model, Xml),
  Model2 = process_changes(Model1),
  persist(Model2).
```

I don't know about you, but this doesn't look composable to me. Instead, it seems fairly noisy and error prone. The temporary variables `Model1` and `Model2` are introduced here only to take the result of one function and feed it to the next.

Of course, you could eliminate the temporary variables and inline the calls:

```
process_xml(Model, Xml) ->
  persist(
    process_changes(
      update(Model, Xml)
    )
  ).
```

This style, known as *staircasing*, is admittedly free of temporary variables, but it's clumsy and hard to read. To understand what goes on here, you have to manually parse it inside-out.

Although Erlang programmers are more or less limited to such clumsy approaches, Elixir gives you an elegant way to chain multiple function calls together:

```
def process_xml(model, xml) do
  model
```

```
    |> update(xml)
    |> process_changes
    |> persist
end
```

The *pipeline* operator `|>` takes the result of the previous expression and feeds it to the next one as the first argument. The resulting code is clean, contains no temporary variables, and reads like the prose, top to bottom, left to right. Under the hood, this code is transformed at compile time to the staircased version. This is again possible because of Elixir's macro system.

The pipeline operator highlights the power of functional programming. You treat functions as data transformations and then combine them in different ways to gain the desired effect.

### 1.2.3. The big picture

There are many other areas where Elixir improves the original Erlang approach. The API for standard libraries is cleaned up and follows some defined conventions. Syntactic sugar is introduced that simplifies typical idioms; and some Erlang data structures, such as the key-value dictionary and set, are rewritten to gain more performance. A concise syntax for working with structured data is provided. String manipulation is improved, and the language has explicit support for Unicode manipulation. In the tooling department, Elixir provides a `mix` tool that simplifies common tasks such as creating applications and libraries, managing dependencies, and compiling and testing code. In addition, a package manager called Hex (https://hex.pm/) is available that makes it simpler to package, distribute, and reuse dependencies.

The list goes on and on; but instead of presenting each feature, I'd like to express a personal sentiment based on my own production experience. Personally, I find it much more pleasant to code in Elixir. The resulting code seems simpler, more readable, and less burdened with boilerplate, noise, and duplication. At the same time, you retain the complete runtime characteristics of pure Erlang code. And you can use all the available libraries from the Erlang ecosystem, both standard and third party.

## 1.3. Disadvantages

No technology is a silver bullet, and Erlang and Elixir are definitely not exceptions. Thus it's worth mentioning some of their shortcomings.

### 1.3.1. Speed

Erlang is by no means the fastest platform out there. If you look at various synthetic benchmarks on the Internet, you usually won't see Erlang high on the list. Erlang programs are run in BEAM and therefore can't achieve the speed of machine-compiled languages, such as C and C++. But this isn't accidental or poor engineering on behalf of the Erlang/OTP team.

The goal of the platform isn't to squeeze out as many requests per seconds as possible, but to keep performance predictable and within limits. The level of performance your Erlang system achieves on a given machine shouldn't degrade significantly, meaning there shouldn't be unexpected system hiccups due to, for example, the garbage collector kicking in. Furthermore, as explained earlier, long-running BEAM processes don't block or significantly impact the rest of the system. Finally, as the load increases, BEAM can use as many hardware resources as possible. If the hardware capacity isn't enough, you can expect graceful system degradation—requests will take longer to process, but the system won't be paralyzed. This is due to the preemptive nature of the BEAM scheduler, which performs frequent context switches that keep the system ticking and favors short-running processes. And of course, you can address higher system demand by adding more hardware.

Nevertheless, intensive CPU computations aren't as performant as, for example, their C/C++ counterparts, so you may consider implementing such tasks in some other language and then integrating the corresponding component into your Erlang system. If most of your system's logic is heavily CPU bound, then you should probably consider some other technology.

### 1.3.2. Ecosystem

The ecosystem built around Erlang isn't small, but it definitely isn't as big as that of some other languages. At the time of writing, a quick search on GitHub reveals just over 11,000 Erlang-based repositories. In contrast, there are more than 600,000 Ruby repositories and almost 1,000,000 for JavaScript. Elixir currently has slightly over 2,000, which is impressive given its young age.

Numbers aside, I have mixed feeling about the available Erlang-based libraries. I can usually find client libraries for what I need (such as MongoDb and 0MQ), but I sometimes have the feeling they aren't mature as I would expect: they either lack proper documentation or don't support all possible features. On the plus side, the community is usually supportive, and I have often seen developers of third-party components provide assistance and extend their libraries if requested.

Regardless, you should be prepared that the choice of libraries won't be as abundant as you may be used to, and in turn you may end up spending extra time on something that would take minutes in other languages. If that happens, keep in mind all the benefits you get from Erlang. As I've explained, Erlang goes a long way toward making it possible to write fault-tolerant systems that can run for a long time with hardly any downtime. This is a big challenge and a specific focus of the Erlang platform. So although it's admittedly unfortunate that the ecosystem isn't as mature as it could be, my sentiment is that Erlang significantly helps with hard problems, even if simple problems can sometimes be more clumsy to solve. Of course, those difficult problems may not always be important. Perhaps you don't expect a high load, or a system doesn't need to run constantly and be extremely fault-tolerant. In such cases, you may want to consider some other technology stack with a more evolved ecosystem.

Elixir's ecosystem currently isn't as mature as Erlang's, but you can use practically any Erlang library from Elixir (with some minor exceptions). Because the language is gaining traction and offers some compelling benefits versus Erlang, this will hopefully improve, and ultimately the Erlang ecosystem will improve as well.

## 1.4. Summary

This chapter defined the purpose and benefits of Erlang and Elixir. There are a couple of points worth remembering:

- Erlang is a technology for developing highly available systems that constantly provide service with little or no downtime. It has been battle tested in diverse large systems for more than two decades.
- Elixir is a modern language that makes development for the Erlang platform much more pleasant. It helps organize code more efficiently and abstracts away boilerplate, noise, and duplication.

Now you can start learning how to develop Elixir-based systems. In the next chapter, you learn about the basic building blocks of Elixir programs.

# Chapter 2. Building blocks

*This chapter covers*

- Using the interactive shell
- Working with variables
- Organizing your code
- Understanding the type system
- Working with operators
- Understanding the runtime

It's time to start learning about Elixir. This chapter presents the basic building blocks of the language, such as modules, functions, and the type system. This is going to be a somewhat long, not particularly exciting tour of language features. But the material presented here is important, because it prepares the stage for exploring more interesting, higher-level topics.

Before starting, make sure you've installed Elixir version 1.0.x (which also requires that you have Erlang version 17.x). There are multiple ways of installing Elixir, and it's best to follow the instructions from the official Elixir site.

With that out of the way, let's start the tour of Elixir. The first thing you should know about is the interactive shell.

**Detailed information**

This book doesn't provide a detailed reference on any of the language or platform features. That would take too much space, and the material would quickly become outdated. Here are some other references you can check out:

- For an alternative syntax quick start, you should look at the Getting Started guide on the Elixir official site: http://elixir-

lang.org/getting_started/1.html.

- A more detailed reference can be found in the online documentation: http://elixir-lang.org/docs/stable/elixir.
- For specific questions, you can turn to elixir-lang mailing list (http://groups.google.com/group/elixir-lang-talk) or the #elixir-lang channel on IRC (irc://irc.freenode.net/elixir-lang).
- Finally, for many things, you'll need to look into the Erlang documentation: http://www.erlang.org/doc. If you're not familiar with Erlang syntax, you may also need to read Elixir's crash course on Erlang (http://elixir-lang.org/crash-course.html).

## 2.1. The interactive shell

The simplest way to experiment and learn about a language's features is through the interactive shell. You can start the Elixir interactive shell from the command line by running the command `iex`:

```
$ iex

Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:8:8]
  [async-threads:10] [kernel-poll:false]

Interactive Elixir (1.0.0) - press Ctrl+C to exit
  (type h() ENTER for help)

iex(1)>
```

Running `iex` starts an instance of the BEAM and then starts an interactive Elixir shell inside it. Runtime information is printed, such as the Erlang and Elixir versions, and then the prompt is provided so you can enter Elixir expressions:

After you type an expression, it's interpreted and executed. Its return value is then printed to the screen.

**Note**

Everything in Elixir is an expression that has a return value. This

includes not only function calls but also constructs like `if` and `case`.

**Tip**

We'll use `iex` extensively throughout the book, especially in the initial chapters. Often, the expression result won't be particularly relevant, and it will be omitted to reduce noise. Regardless, keep in mind that each expression returns a result, and when you enter an expression in the shell, its result will be presented.

You can type practically anything that constitutes valid Elixir code, including more complicated multiline expressions:

Notice how the shell doesn't evaluate the expression until you finish it on the last line. In Elixir, you need no special characters, such as semicolons, to indicate the end of an expression. Instead, a line break indicates the end of an expression, if the expression is complete. Otherwise, the parser waits for additional input until the expression becomes complete.

You can put multiple expressions on a single line by separating them with a semicolon:

Keep in mind that these rules hold not only in the shell, but also in standard Elixir code. As mentioned, whatever works in the source code usually works in the shell.

The quickest way to leave the shell is to press Ctrl + C twice. Doing so brutally kills the OS process and all background jobs that are executing. Because the shell is mostly used for experimenting and shouldn't be used to run real production systems, it's usually fine to

terminate it this way. If you want a more polite way of stopping the system, you can invoke `System.halt`.

**Note**

There are multiple ways to start Elixir and Erlang runtime and running programs. You'll learn a bit about all of them by the end of the chapter. In the first part of this book you'll mostly work with the `iex` shell, because it's a simple and efficient way of experimenting with the language.

You can do many things with the shell, but most often you'll use this basic feature of entering expressions and inspecting their results. You can research for yourself what else can be done in the shell. Basic help can be obtained with the `h` command:

```
iex(4)> h
```

Entering this in the shell will output an entire screen of `iex` related instructions. You can also look for the documentation of the `IEx` module, which is responsible for the shell's workings:

```
iex(5)> h IEx
```

You can find the same help in the online documentation at http://elixir-lang.org/docs/stable/iex.

Now that you have a basic tool with which to experiment, let's research the features of the language. We'll start with variables.

## 2.2. Working with variables

Elixir is a *dynamic* programming language, which means you don't explicitly declare a variable or its type. Instead, the variable type is determined by whatever data it contains at the moment. In Elixir terms, assignment is called *binding*. When you initialize a variable with a value, the variable is *bound* to that value:

```
iex(1)> monthly_salary = 10000
10000
```

Binds a variable

Result of the last expression

Each expression in Elixir has a result. In the case of the = operator, the result is whatever is on the right side of the operator. After the expression is evaluated, the shell prints this result to the screen.

Now you can reference the variable:

```
iex(2)> monthly_salary
10000
```

Expression that returns the value of the variable

Value of the variable

The variable can of course be used in complex expressions:

```
iex(3)> monthly_salary * 12
120000
```

In Elixir, a variable name always starts with a lowercase alphabetic character or an underscore. After that, any combination of alphanumerics and underscores is allowed. The prevalent convention is to use only lowercase letters, digits, and underscores:

```
valid_variable_name
also_valid_1

validButNotRecommended

NotValid
```

As added syntactic sugar, variable names can end with the characters `?` and `!`:

```
valid_name?
also_ok!
```

This convention is more often used when naming functions.

Data in Elixir is immutable: its contents can't be changed. But variables can be rebound to a different value:

```
iex(1)> monthly_salary = 10000      <-- Sets the initial value
10000
iex(2)> monthly_salary              <-- Verifies it
10000
iex(3)> monthly_salary = 11000      <-- Rebinds the initial value
11000
iex(4)> monthly_salary              <-- Verifies the effect of rebinding
11000
```

Rebinding doesn't mutate the existing memory location. It reserves new memory and reassigns the symbolic name to the new location.

**Note**

You should always be aware of the fact that data is immutable. Once a memory location is occupied with data, it can't be modified until it's released. But variables can be rebound, which makes them point to a different memory location. Thus variables are mutable, whereas the data they point to is immutable.

Elixir is a garbage-collectable language, which means you don't have to manually release memory. When a variable goes out of scope, the corresponding memory is eligible for garbage collection and will be released sometime in the future, when the garbage collector cleans up the memory.

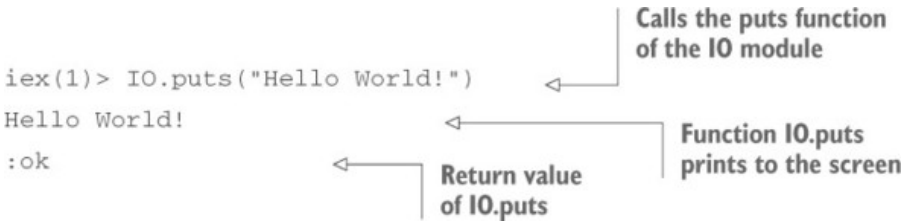
## 2.3. Organizing your code

Being a functional language, Elixir relies heavily on functions. Due to the immutable nature of the data, a typical Elixir program consists of many small functions. You'll witness this in chapters 3 and 4, as you start using some typical functional idioms. Multiple functions can be further organized into modules.

### 2.3.1. Modules

A *module* is a collection of functions, something like a namespace. Every Elixir function must be defined inside a module.

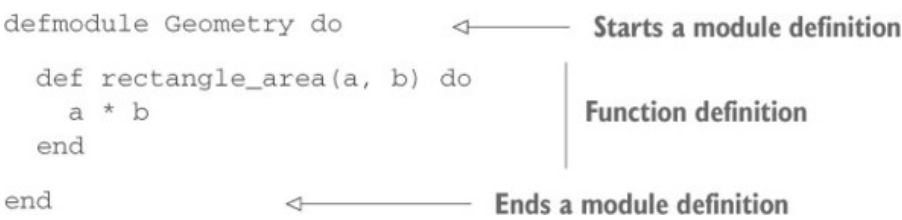
Elixir comes with a standard library that provides many useful modules. For example, the `IO` module can be used to do various I/O operations. The function `puts` from the `IO` module can be used to print a message to the screen:


Calls the puts function of the IO module
iex(1)> IO.puts("Hello World!")
Hello World!
:ok
Function IO.puts prints to the screen
Return value of IO.puts

As you can see in the example, to call a function of a module, you use the syntax `ModuleName.function_name(args)`.

To define your own module, you use the `defmodule` construct. Inside the module, you define functions using the `def` construct. The following listing demonstrates the definition of a module.

**Listing 2.1. Defining a module (geometry.ex)**




There are two ways you can use this module. You can copy/paste this definition directly into `iex`—as mentioned, almost anything can be typed into the shell. Another way is to tell `iex` to interpret the file while starting:

```
$ iex geometry.ex
```

Regardless of which method you choose, the effect is the same. The code is compiled, and the resulting module is loaded into the runtime and can be used from the shell session. Let's try it:

```
$ iex geometry.ex

iex(1)> Geometry.rectangle_area(6, 7)
42
```

Invokes the function

Function result

That was simple! You created a `Geometry` module, loaded it into a shell session, and used it to compute the area of a rectangle.

**Note**

As you may have noticed, the filename has the .ex extension. This is a common convention for Elixir source files.

In the source code, a module must be defined in a single file. A single file may contain multiple module definitions:

```
defmodule Module1 do
  ...
end

defmodule Module2 do
  ...
end
```

A module name must follow certain rules. It starts with an uppercase letter and is usually written in *CamelCase* style. A module name can consist of alphanumerics, underscores, and the dot (`.`) character. The latter is often used to organize modules hierarchically:

```
defmodule Geometry.Rectangle do
  ...
end

defmodule Geometry.Circle do
  ...
end
```

The dot character is a convenience. Once the code is compiled, there are no special hierarchical relations between modules, nor are there services to query the hierarchy. It's just syntactical sugar that can help you scope your names.

You can also nest modules:

```
defmodule Geometry do
```

```
  defmodule Rectangle do
    ...
  end
  ...
end
```

The child module can be referenced with `Geometry.Rectangle`. Again, this nesting is a convenience. After the compilation, there is no special relation between the modules `Geometry` and `Geometry.Rectangle`.

### 2.3.2. Functions

A function must always be a part of a module. Function names follow the same conventions as variables: they start with a lowercase letter or underscore character, followed by a combination of alphanumerics and underscores.

As with variables, function names can end with the characters `?` and `!`. The `?` character is often used to indicate a function that returns either *true* or *false*. Placing the character `!` at the end of the name indicates a function that may raise a runtime error. Both of these are conventions, rather than rules, but it's best to follow them and respect the community style.

Functions can be defined using the `def` construct:

The definition starts with the `def` construct, followed by the function name, the argument list, and the body enclosed in a `do...end` block. Because you're dealing with a dynamic language, there are no type specifications for arguments.

**Note**

Notice that `defmodule` and `def` aren't referred to as keywords.

That's because they're not! Instead, these are compilation constructs called *macros*. You don't need to worry about how this works; it's explained a bit later in this chapter. If it helps, you can think of `def` and `defmodule` as keywords, but be aware that this isn't exactly true.

If a function has no arguments, you can omit parentheses:

```
defmodule Program do
  def run do
    ...
  end
end
```

What about the return value? Recall that in Elixir, everything that has a return value is an expression. The return value of a function is the return value of its last expression. There is no explicit return in Elixir.

**Note**

Given that there is no explicit return, you might wonder how complex functions work. This will be covered in detail in chapter 3, where you'll learn about branching and conditional logic. The general rule is to keep functions short and simple, which makes it easy to compute the result and return it from the last expression.

You saw an example of returning a value in listing 2.1, but let's repeat it:

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end
```

Calculates the area and returns the result

You can now verify this. Start the shell again, and then try the `rectangle_area` function:

```
$ iex geometry.ex

iex(1)> Geometry.rectangle_area(3, 2)
6
```

Calls the function

Function return value

If a function definition is short, you can use a condensed form and define it in a single line:

```
defmodule Geometry do
  def rectangle_area(a, b), do: a * b
end
```

To call a function defined in another module, you use the module name followed by the function name:

```
iex(1)> Geometry.rectangle_area(3, 2)
6
```

Of course, you can always store the function result to a variable:

```
iex(2)> area = Geometry.rectangle_area(3, 2)
6

iex(3)> area
6
```

Calls the function and stores its result

Verifies the variable content

Parentheses are optional in Elixir, so you can omit them:

```
iex(4)> Geometry.rectangle_area 3, 2
6
```

Personally, I find that omitting parentheses makes the code ambiguous, so my advice is to always include them when calling a function. The exception is a function that has no arguments. In this case, parentheses only increase the noise, so it's better to omit them.

If a function resides in the same module, you can omit the module prefix:

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end

  def square_area(a) do
    rectangle_area(a, a)
  end
end
```

Call to a function in the same module

Given that Elixir is a functional language, you often need to combine functions, passing the result of one function as the argument to the next one. Elixir comes with a built-in operator |>, called the *pipeline* operator, that does exactly this:

```
iex(5)> -5 |> abs |> Integer.to_string |> IO.puts
5
```

This code is transformed at *compile time* into the following:

```
iex(6)> IO.puts(Integer.to_string(abs(-5)))
5
```

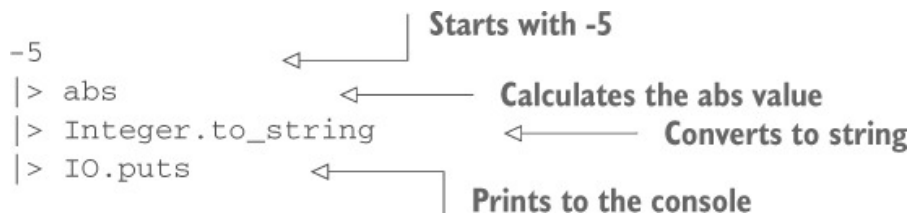
More generally, the pipeline operator places the result of the previous call as the first argument of the next call. So the following code

```
prev(arg1, arg2) |> next(arg3, arg4)
```

is translated at compile time to

```
next(prev(arg1, arg2), arg3, arg4)
```

Arguably, the pipeline version is more readable because the sequence of execution is read from left to right. The pipeline operator looks especially elegant in source files, where you can lay out the pipeline over multiple lines, which makes the code read like prose:




**Note**

Multiline pipelines don't work in the shell. The previous code is an example of a syntax that works only in the source file and can't be entered directly in the shell. When you paste this code into the shell, it will raise an error. The reason is that the first line is a proper standalone Elixir expression, and the shell will execute it immediately. The rest of the code isn't syntactically correct, because it starts with the |> operator.

### 2.3.3. Function arity

Arity is a fancy name for the number of arguments a function receives. A function is uniquely identified by its containing module, its name, and its arity. Take a look at the following function:

```
defmodule Rectangle do
  def area(a, b) do          <-- Function with two arguments
    ...
  end
end
```

The function `Rectangle.area` receives two arguments, so it's said to be a function of arity 2. In the Elixir world, this function is often called `Rectangle.area/2`, where the `/2` part denotes the function's arity.

Why is this important? Because two functions with the same name but different arities are two different functions, as the following example demonstrates.

**Listing 2.2. Functions with the same name but different arities (arity_demo.ex)**

```
defmodule Rectangle do
  def area(a), do: area(a, a)      <-- Rectangle.area/1

  def area(a, b), do: a * b        <-- Rectangle.area/2
end
```

Load this module to the shell, and try the following:

```
iex(1)> Rectangle.area(5)
25

iex(2)> Rectangle.area(5,6)
30
```

As you can see, these two functions act completely differently. The name might be overloaded, but the arities differ, so we talk about them as two distinct functions, each having its own implementation.

It usually makes no sense for different functions with a same name to have completely different implementations. More commonly, a lower-

arity function delegates to a higher-arity function, providing some default arguments. This is what happens in listing 2.2, where `Rectangle.area/1` delegates to `Rectangle.area/2`.

Let's look at another example.

**Listing 2.3. Same-name functions, different arities, default params (arity_calc.ex)**

```
defmodule Calculator do
  def sum(a) do
    sum(a, 0)
  end

  def sum(a, b) do

    a + b
  end
end
```

Calculator.sum/1 delegates to Calculator.sum/2

Calculator.sum/2 contains the implementation

Again, a lower-arity function is implemented in terms of a higher-arity one. This pattern is so frequent that Elixir allows you to specify defaults for arguments by using the `\\` operator followed by the argument's default value:

```
defmodule Calculator do
  def sum(a, b \\ 0) do
    a + b
  end
end
```

Defining a default value for argument b

This definition generates two functions exactly like in listing 2.3.

You can set the defaults for any combination of arguments:

```
defmodule MyModule do
  def fun(a, b \\ 1, c, d \\ 2) do
    a + b + c + d
  end
end
```

Setting defaults for multiple arguments

Always keep in mind that default values generate multiple functions of

the same name with different arities. So the previous code generates three functions: `MyModule.fun/2`, `MyModule.fun/3`, and `MyModule.fun/4`.

Because arity distinguishes multiple functions of the same name, it's not possible to have a function accept a variable number of arguments. There is no counterpart of C's `...` or JavaScript's `arguments`.

### 2.3.4. Function visibility

When you define a function using the `def` macro, the function is made public: it can be called by anyone else. In Elixir terminology, it's said that the function is *exported*. You can also use the `defp` macro to make the function private. A private function can be used only inside the module it's defined in. The following example demonstrates.

**Listing 2.4. Module with a public and a private function (private_fun.ex)**

```
defmodule TestPrivate do
  def double(a) do
    sum(a, a)
  end

  defp sum(a, b) do
    a + b
  end
end
```

Public function

Calls the private function

Private function

The module `TestPrivate` defines two functions. The function `double` is exported and can be called from outside. Internally, it relies on the private function `sum` to do its work.

Let's try this in the shell. Load the module, and do the following:

```
iex(1)> TestPrivate.double(3)
6
```

That works. Now let's try calling the private function:

```
iex(2)> TestPrivate.sum(3, 4)
** (UndefinedFunctionError) undefined function: TestPrivate.sum/2
...
```

As you can see, the private function can't be invoked outside the module.

### 2.3.5. Imports and aliases

Calling functions from another module can sometimes be cumbersome, because you need to reference the module name. If your module often calls functions from another module, you can *import* that other module into your own. Importing a module allows you to call its public functions without prefixing them with the module name:

```
defmodule MyModule do
  import IO                    <-- Imports the module

  def my_function do
    puts "Calling imported function."    <-- You can use puts instead of IO.puts
  end
end
```

Of course, you can import multiple modules. In fact, the standard library's `Kernel` module is automatically imported into every module. This is because `Kernel` contains functions that are often used, so automatic importing makes their usage easier.[1].

1 You can check out what functions are available in the Kernel module in the online documentation at http://elixir-lang.org/docs/stable/elixir/Kernel.html.

An alternative to `import` is `alias`, a construct that makes it possible to reference a module under a different name:

```
defmodule MyModule do
  alias IO, as: MyIO           <-- Creates an alias for IO

  def my_function do
    MyIO.puts("Calling imported function.")    <-- Calls a function using the alias
  end
end
```

Aliases can be useful if a module has a long name. For example, if your application is heavily divided into a deeper module hierarchy, it can be cumbersome to reference modules via fully qualified names.

Aliases can help with this. For example, let's say you have a `Geometry.Rectangle` module. You can alias it in your client module and use a shorter name:

```
defmodule MyModule do
  alias Geometry.Rectangle, as: Rectangle   <-- Sets up an alias to a module

  def my_function do
    Rectangle.area(...)                     <-- Calls a module function using the alias
  end
end
```

As the result, the code in the client module is more compact and arguably easier to read. Personally, I tend to avoid aliases, because they increase ambiguity. But in some cases they can improve readability, especially if you call functions from a long-named module many times.

### 2.3.6. Module attributes

The purpose of module attributes is twofold: they can be used as compile-time constants, and you can *register* any attribute, which can be then queried in runtime. Let's look at an example. The following module provides basic functions for working with circles:

```
iex(1)> defmodule Circle do
          @pi 3.14159                   <-- Defines a module attribute

          def area(r), do: r*r*@pi                  | Uses a
          def circumference(r), do: 2*r*@pi         | module attribute
        end

iex(2)> Circle.area(1)
3.14159

iex(3)> Circle.circumference(1)
6.28318
```

Notice how you define a module directly in the shell. This is permitted and makes it possible to experiment without storing any files on the disk.

The important thing about the `@pi` constant is that it exists only during the compilation of the module, when the references to it are inlined.

Moreover, an attribute can be *registered*, which means it will be stored in the generated binary and can be accessed in runtime. Elixir registers some module attributes by default. For example, the attributes `@moduledoc` and `@doc` can be used to provide documentation for modules and functions:

```
defmodule Circle do
  @moduledoc "Implements basic circle functions"

  @pi 3.14159

  @doc "Computes the area of a circle"
  def area(r), do: r*r*@pi

  @doc "Computes the circumference of a circle"
  def circumference(r), do: 2*r*@pi
end
```

To try this, however, you need to generate a compiled file. Here's a quick way to do it. Save this code to the circle.ex file somewhere, and then run `elixirc circle.ex`. This will generate the file Elixir.Circle.beam. Next, start `iex` shell from the same folder. You can now retrieve the attribute at runtime:

```
iex(1)> Code.get_docs(Circle, :moduledoc)
{1, "Implements basic circle functions"}
```

More interesting is that other tools from the Elixir ecosystem know how to work with these attributes. For example, you can use the help feature of `iex` to see the module's documentation:

```
iex(2)> h Circle                    <-------- Module documentation

                                        Circle

Implements basic circle functions

iex(3)> h Circle.area               <-------- Function documentation

                                    def area(r)

Computes the area of a circle
```

Furthermore, you can use the `ex_doc` tool (see https://github.com/elixir-lang/ex_doc) to generate HTML documentation for your project. This is the way Elixir documentation is produced, and if you plan to build more complex projects, especially something that will be used by many different clients, you should consider using `@moduledoc` and `@doc`.

The underlying point is that registered attributes can be used to attach meta information to a module, which can then be used by other Elixir (and even Erlang) tools. There are many other preregistered attributes, and you can also register your own, custom attributes. Take a look at the documentation for the module `Module` (http://elixir-lang.org/docs/stable/elixir/Module.html) for more details.

#### Type specifications

Type specifications (often called typespecs) are another important feature based on attributes. This feature allows you to provide type information for your functions, which can later be analyzed with a static analysis tool called `dialyzer` (www.erlang.org/doc/man/dialyzer.html).

Let's extend the `Circle` module to include typespecs:

```
defmodule Circle do
  @pi 3.14159

  @spec area(number) :: number      <-- Type specification for area/1
  def area(r), do: r*r*@pi

  @spec circumference(number) :: number   <-- Type specification for circumference/1
  def circumference(r), do: 2*r*@pi
end
```

Here you use the `@spec` attribute to indicate that both functions accept and return a number.

Typespecs provide a way of compensating for the lack of a static type system. This can be useful in conjunction with the `dialyzer` tool to perform static analysis of your programs. Furthermore, typespecs allow you to document your functions better. Remember that Elixir is a dynamic language, so function inputs and output can't be easily deduced by looking at the function's signature. Typespecs can help significantly with this, and I can personally attest that it's much easier to understand someone else's code when typespecs are provided.

For example, look at the typespec for the Elixir function `List.insert_at/3`:

```
@spec insert_at(list, integer, any) :: list
```

Even without looking at the code or reading the docs, you can reasonably guess that this function inserts a term of any type (third argument) to a list (first argument) at a given position (second argument) and returns a new list.

We won't be using typespecs in this book, mostly to keep the code as short as possible. But if you plan to build more complex systems, my advice is to seriously consider using typespecs. You can find a detailed reference in the official docs at http://elixir-lang.org/docs/stable/elixir/Kernel.Typespec.html.

### 2.3.7. Comments

Comments in Elixir start with the character `#`, which indicates that the rest of the line is a comment. Block comments aren't supported. If you need to comment multiple lines, prefix each one with a `#` character:

```
# This is a comment
a = 3.14      # so is this
```

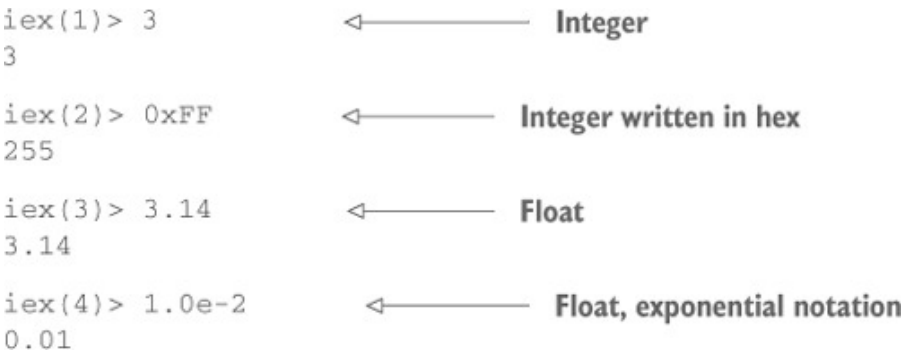
At this point, we're done with the basics of functions and modules. You're aware of the primary code-organization techniques. With this out of our way, it's time to look at the Elixir type system.

## 2.4. Understanding the type system

At its core, Elixir uses the Erlang type system. Consequently, integration with Erlang libraries is usually simple. The type system itself is reasonably simple, but if you're coming from a classical OO language, you'll find it significantly different from what you're used to. This section covers basic Elixir types and also discusses some implications of immutability. To begin, let's look at numbers.

### 2.4.1. Numbers

Numbers can be integers or floats, and they work mostly as you would expect:




Standard arithmetic operators are supported:

```
iex(5)> 1 + 2 * 3
7
```

The division operator `/` works differently than you might expect. It always returns a float value:

```
iex(6)> 4/2
2.0

iex(7)> 3/2
1.5
```

To perform integer division or to calculate the remainder, you can use auto-imported `Kernel` functions:

```
iex(8)> div(5,2)
2

iex(9)> rem(5,2)
1
```

As added syntactic sugar, you can use the underscore character as a visual delimiter:

```
iex(10)> 1_000_000
1000000
```

There is no upper limit on the integer size, and you can use arbitrarily

large numbers:

```
iex(11)>
99999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999999999
```

If you're worried about memory size, it's best to consult the official Erlang memory guide at http://mng.bz/Pspr. An integer takes up as much space as needed to accommodate the number, whereas a float occupies either 32 or 64 bits, depending on the build architecture of the virtual machine. Floats are internally represented in IEEE 754-1985 (binary precision) format.

### 2.4.2. Atoms

Atoms are literal named constants. They're similar to symbols in Ruby or enumerations in C/C++. Atom constants start with a colon character, followed by a combination of alphanumerics and/or underscore characters:

```
:an_atom
:another_atom
```

It's possible to use spaces in the atom name with the following syntax:

```
:"an atom with spaces"
```

An atom consists of two parts: the *text* and the *value*. The atom text is whatever you put after the colon character. At runtime, this text is kept in the *atom table*. The value is the data that goes into the variable, and it's merely a reference to the atom table.

This is exactly why atoms are best used for named constants. They're efficient both memory- and performance-wise. When you say

```
variable = :some_atom
```

the variable doesn't contain the entire text, but only a reference to the atom table. Therefore, memory consumption is low, the comparisons are fast, and the code is still readable.

#### Aliases

There is another syntax for atom constants. You can omit the beginning colon, and start with uppercase:

```
AnAtom
```

This is called an *alias*, and at compile time it's transformed into `:"Elixir.AnAtom"`:

```
iex(1)> AnAtom == :"Elixir.AnAtom"
true
```

When you use an alias, the compiler implicitly adds the *Elixir.* prefix to its text and inserts an atom there. But if an alias already contains that prefix, it's not added. Consequently, the following also works:

```
iex(2)> AnAtom == Elixir.AnAtom
true
```

You may recall from earlier that you can also use aliases to give alternate names to modules:

```
iex(3)> alias IO, as: MyIO

iex(4)> MyIO.puts("Hello!")
Hello!
```

It's no accident that the term *alias* is used for both things. When you write `alias IO, as: MyIO`, you instruct the compiler to transform `MyIO` into `IO`. Resolving this further, the final result emitted in the generated binary is `:Elixir.IO`. Therefore, with an alias set up, the following also holds:

```
iex(5)> MyIO == Elixir.IO
true
```

All of this may seem strange, but it has an important underlying purpose. Aliases support the proper resolution of modules. This will be discussed at the end of the chapter when we revisit modules and look at how they're loaded at runtime.

#### Atoms as booleans

It may come as a surprise that Elixir doesn't have a dedicated boolean type. Instead, the atoms `:true` and `:false` are used. As syntactic sugar, Elixir allows you to reference these atoms without the starting colon character:

```
iex(1)> :true == true
true

iex(2)> :false == false
true
```

The term *boolean* is still used in Elixir to denote an atom that has a value of either `:true` or `:false`. The standard logical operators work with boolean atoms:

```
iex(1)> true and false
false

iex(2)> false or true
true

iex(3)> not false
true

iex(4)> not :an_atom_other_than_true_or_false
** (ArgumentError) argument error
```

Always keep in mind that a boolean is just an atom that has a value of true or false.

#### Nil and truthy values

Another special atom is `:nil`, which works somewhat similarly to `null` from other languages. You can reference `nil` without a colon:

```
iex(1)> nil == :nil
true
```

The atom `nil` plays a role in Elixir's additional support for *truthness*, which works similarly to the way it's used in mainstream languages such as C/C++ and Ruby. The atoms `nil` and `false` are treated as *falsy* values, whereas everything else is treated as a *truthy* value.

This property can be used with Elixir's *short-circuit* operators `||`, `&&` and `!`. The operator `||` returns the first expression that isn't falsy:

```
iex(1)> nil || false || 5 || true
5
```

Because both `nil`, and `false` are falsy expressions, the number 5 is returned. Notice that subsequent expressions won't be evaluated at all. If all expressions evaluate to a falsy value, then the result of the last expression is returned.

The operator `&&` returns the second expression, but only if the first expression is truthy. Otherwise, it returns the first expression without evaluating the second one:

```
iex(1)> true && 5
5

iex(2)> false && 5
false

iex(3)> nil && 5
nil
```

Short-circuiting can be used for elegant operation chaining. For example, if you need to fetch a value from cache, a local disk, or a remote database, you can do something like this:

```
read_cached || read_from_disk || read_from_database
```

Similarly, you can use the operator `&&` to ensure that certain conditions are met:

```
database_value = connection_established? && read_data
```

In both examples, short-circuit operators make it possible to write concise code without resorting to complicated nested conditional constructs.

### 2.4.3. Tuples

Tuples are something like untyped structures, or records, and they're most often used to group a *fixed number* of elements together. The following snippet defines a tuple consisting of person's name and age:

```
iex(1)> person = {"Bob", 25}
{"Bob", 25}
```

To extract an element from the tuple, you can use the `Kernel.elem/2` function, which accepts a tuple and the zero-based index of the element. Recall that the `Kernel` module is auto-imported, so you can call `elem` instead of `Kernel.elem`:

```
iex(2)> age = elem(person, 1)
25
```

To modify an element of the tuple, you can use the `Kernel.put_elem/3` function, which accepts a tuple, a zero-based index, and the new value of the field in the given position:

```
iex(3)> put_elem(person, 1, 26)
{"Bob", 26}
```

The function `put_elem` doesn't modify the tuple. It returns the new version, keeping the old one intact. Recall that data in Elixir is immutable, and you can't do an in-memory modification of a value. You can verify that the previous call to `put_elem` didn't change the `person` variable:

```
iex(4)> person
{"Bob", 25}
```

So how can you use the `put_elem` function, then? You need to store its result to another variable:

```
iex(5)> older_person = put_elem(person, 1, 26)
{"Bob", 26}

iex(6)> older_person
{"Bob", 26}
```

Recall that variables can be rebound, so you can also do the following:

```
iex(7)> person = put_elem(person, 1, 26)
{"Bob", 26}
```

By doing this, you have effectively rebound the `person` variable to the new memory location. The old location isn't referenced by any

other variable, so it's eligible for garbage collection.

**Note**

You may wonder if this approach is memory efficient. In most cases, there will be little data copying, and the two variables will share as much memory as possible. This is explained later in this section, when we discuss immutability.

Tuples are most appropriate to group a small, fixed number of elements together. When you need a dynamically sized collection, you can use *lists*.

### 2.4.4. Lists

Lists in Erlang are used to manage dynamic, *variable-sized* collections of data. The syntax deceptively resembles arrays from other languages:

```
iex(1)> prime_numbers = [1, 2, 3, 5, 7]
[1, 2, 3, 5, 7]
```

Lists may look like arrays, but they work like singly linked lists. To do something with the list, you have to traverse it. Therefore, most of the operations on lists have an O(n) complexity, including the `Kernel.length/1` function, which iterates through the entire list to calculate its length:

```
iex(2)> length(prime_numbers)
5
```

#### List utility functions

There are many operations you can do with lists, but this section mentions only a couple of the most basic ones. For a detailed reference, see the documentation for the `List` module (http://elixir-lang.org/docs/stable/elixir/List.html). There are also many helpful services in the `Enum` module (http://elixir-lang.org/docs/stable/elixir/Enum.html).

The `Enum` module deals with many different enumerable structures and is not limited to lists. The concept of enumerables is explained in details in chapter 4, when we discuss protocols.

To get an element of a list, you can use the `Enum.at/2` function:

```
iex(3)> Enum.at(prime_numbers, 4)
7
```

`Enum.at` is again an O(n) operation: it iterates from the beginning of the list to the desired element. Lists are never a good fit when direct access is called for. For those purposes, either tuples, maps, or a higher-level data structure is appropriate.

You can check whether a list contains a particular element with the help of the `in` operator:

```
iex(4)> 5 in prime_numbers
true

iex(5)> 4 in prime_numbers
false
```

To manipulate lists, you can use functions from the `List` module. For example, `List.replace_at/3` modifies the element at a certain position:

```
iex(6)> List.replace_at(prime_numbers, 0, 11)
[11, 2, 3, 5, 7]
```

As was the case with tuples, the modifier doesn't mutate the variable, but returns the modified version of it, which you need to store to another variable:

```
iex(7)> new_primes = List.replace_at(prime_numbers, 0, 11)
[11, 2, 3, 5, 7]
```

Or you can rebind to the same one:

```
iex(8)> prime_numbers = List.replace_at(prime_numbers, 0, 11)
[11, 2, 3, 5, 7]
```

You can insert a new element at the specified position with the `List.insert_at` function:

```
iex(9)> List.insert_at(prime_numbers, 4, 1)
[11, 2, 3, 5, 1, 7]
```

Inserts a new element at the fifth position.

To append to the end, you can use a negative value for the insert position:

```
iex(10)> List.insert_at(prime_numbers, -1, 1)
[11, 2, 3, 5, 7, 1]
```

Value of -1 indicates that the element should be appended to the end of the list

Like most list operations, modifying a random element has a complexity of O(n). In particular, appending to the end is expensive because it always take *n* steps, *n* being the length of the list.

In addition, the dedicated operator `++` is available. It concatenates two lists:

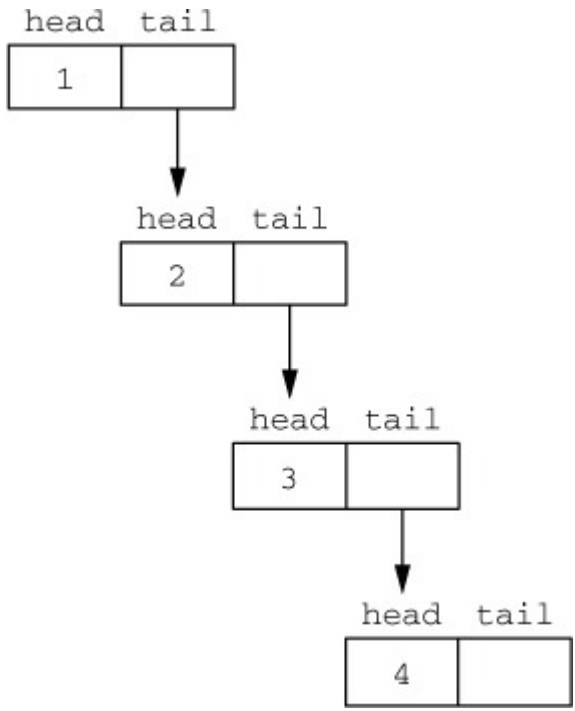
```
iex(11)> [1,2,3] ++ [4,5]
[1, 2, 3, 4, 5]
```

Again, the complexity is O(n), *n* being the length of the left list (the one you're appending to). In general, you should avoid adding elements to the end of a list. Lists are most efficient when new elements are pushed to the top, or popped from it. To understand why, let's look at the recursive nature of lists.

**Recursive list definition**

An alternative way of looking at lists is to think of them as recursive structures. A list can be represented by a pair (*head*, *tail*), where *head* is the first element of the list and *tail* "points" to the (*head*, *tail*) pair of the remaining elements, as illustrated in figure 2.1.

**Figure 2.1. Recursive structure of the list [1, 2, 3, 4]**




If you're familiar with Lisp, then you know this concept as cons cells. In Elixir, there is a special syntax to support recursive list definition:

```
a_list = [head | tail]
```

`head` can be any type of data, whereas `tail` is itself a list. If `tail` is an empty list, it indicates the end of the entire list. Let's look at some examples:

```
iex(1)> [1 | []]
[1]

iex(2)> [1 | [2 | []]]
[1, 2]

iex(3)> [1 | [2]]
[1, 2]

iex(4)> [1 | [2, 3, 4]]
[1, 2, 3, 4]
```

This is just another syntactical way of defining lists, but it illustrates what a list is. It's a pair with two values: a head and a tail, the tail being itself a list. The following snippet is a canonical recursive

definition of a list:

```
iex(1)> [1 | [2 | [3 | [4 | []]]]]
[1, 2, 3, 4]
```

Of course, nobody wants to write constructs like this one. But it's important that you're always aware that internally, lists are recursive structures of (*head*, *tail*) pairs.

To get the head of the list, you can use the `hd` function. The tail can be obtained by calling the `tl` function:

```
iex(1)> hd([1, 2, 3, 4])
1

iex(2)> tl([1, 2, 3, 4])
[2, 3, 4]
```

Both operations are O(1), because they amount to reading one or the other value from the (*head*, *tail*) pair.

**Note**

For the sake of completeness, it should be mentioned that the tail doesn't need to be a list. It can be any type. When the tail isn't a list, it's said that the list is *improper*, and most of the standard list manipulations won't work. Improper lists have some special usages, but we won't deal with them in this book.

Knowing the recursive nature of the list, it's simple and efficient to push a new element to the top of the list:

```
iex(1)> a_list = [5, :value, true]
[5, :value, true]

iex(2)> new_list = [:new_element | a_list]
[:new_element, 5, :value, true]
```

Construction of the `new_list` is an O(1) operation, and no memory copying occurs—the tail of the `new_list` *is* the `a_list`! To understand how this works, let's discuss the internal details of immutability a bit.

### 2.4.5. Immutability

As has been mentioned before, Elixir data can't be mutated. Every function returns the new, modified version of the input data. You have to take the new version into another variable, or rebind it to the same symbolic name. In any case, the result resides in another memory location. The modification of the input will result in some data copying, but generally, most of the memory will be shared between the old and the new version. Let's take a closer look at how this works.

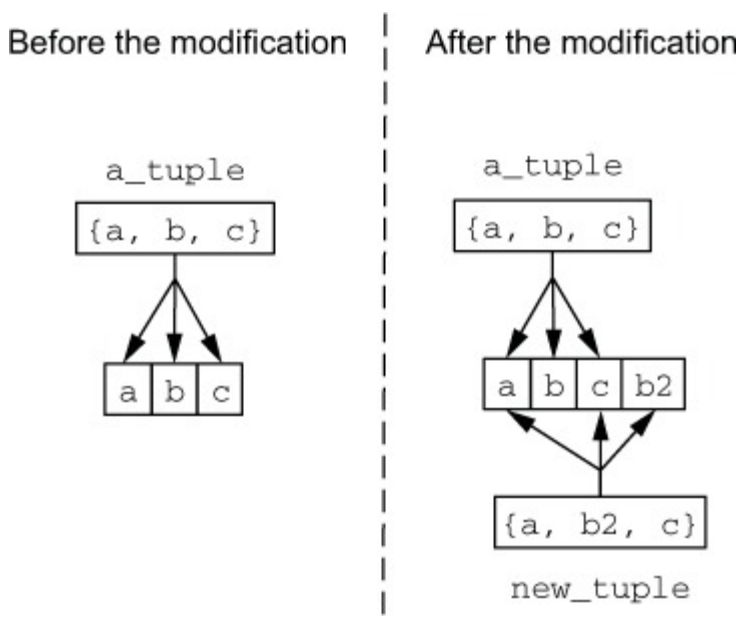
#### Modifying tuples

Let's start with tuples. The modified tuple is always a complete, shallow copy of the old version. Consider the following code:

```
a_tuple = {a, b, c}
new_tuple = put_elem(a_tuple, 1, b2)
```

The variable `new_tuple` will contain a shallow copy of `a_tuple`, differing only in the second element, as illustrated in figure 2.2.

**Figure 2.2. Modifying a tuple creates a shallow copy of it.**



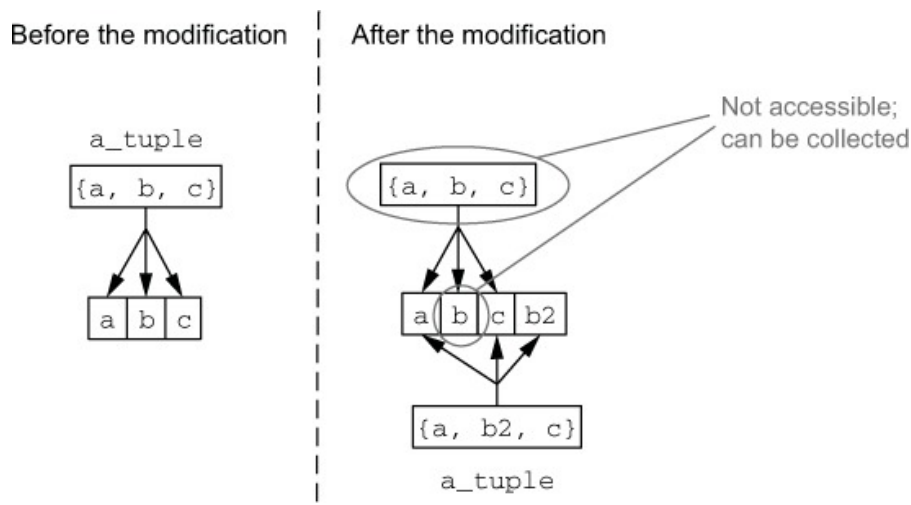
Both tuples reference variables `a` and `c`, and whatever is in those variables is shared (and not duplicated) between both tuples. So `new_tuple` is a shallow copy of the original `a_tuple`.

What happens if you rebind a variable?

```
a_tuple = {a, b, c}
a_tuple = put_elem(a_tuple, 1, b2)
```

In this case, after rebinding, the variable `a_tuple` references another memory location. The old location of `a_tuple` isn't accessible and is available for garbage collection. The same holds for the variable `b` referenced by the old version of the tuple, as illustrated in figure 2.3.

**Figure 2.3. Rebinding a tuple makes the old data garbage-collectible.**

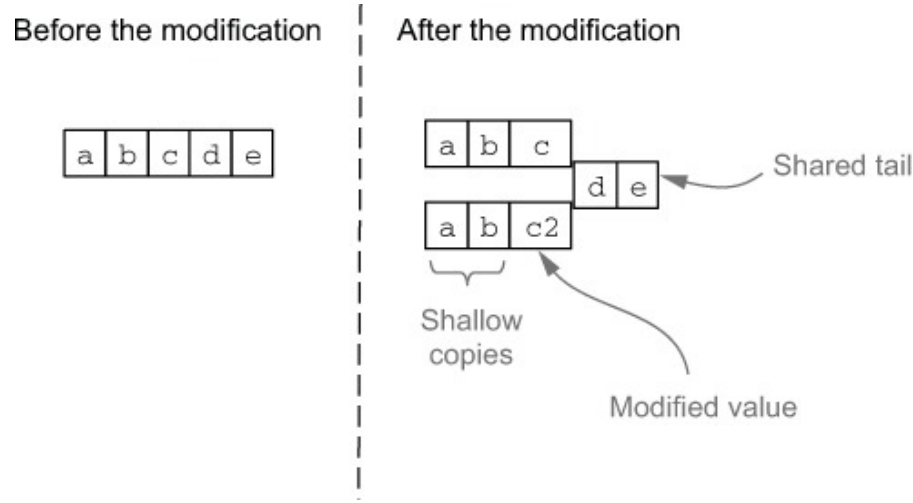



Keep in mind that tuples are always copied, but the copying is shallow. Lists, however have different properties.

#### Modifying lists

When you modify the *n*th element of a list, the new version will contain shallow copies of the first $n - 1$ elements, followed by the modified element. After that, the tails are completely shared, as illustrated in figure 2.4.

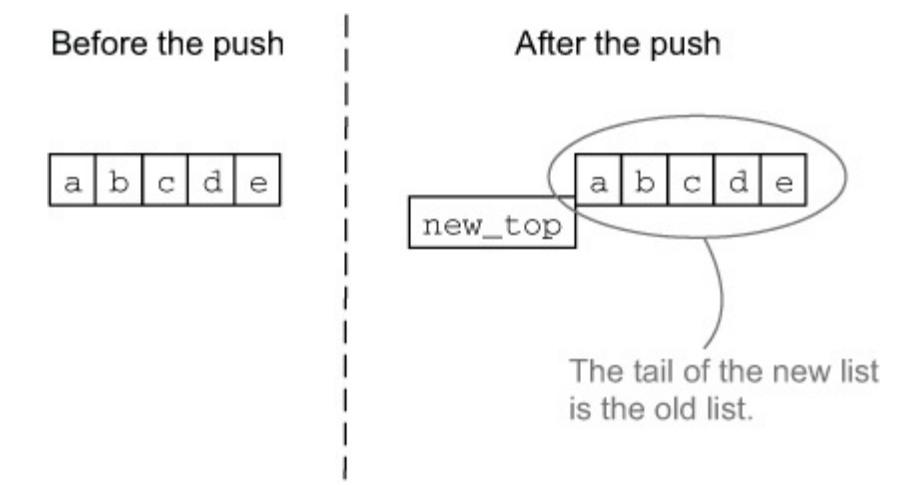
**Figure 2.4. Modifying a list**




This is precisely why adding elements to the end of a list is expensive. To append a new element at the tail, you have to iterate and (shallow) copy the entire list!

In contrast, pushing an element to the top of a list doesn't copy anything, which makes it the least expensive operation, as illustrated in figure 2.5. In this case, the new list's tail *is* the previous list. This is often used in Elixir programs when iteratively building lists. In such cases, it's best to push consecutive elements to the top and then, after the list is constructed, reverse the entire list in a single pass

**Figure 2.5. Pushing a new element to the top of the list**



**Benefits**

Immutability may seem strange, and you may wonder about its purpose. There are two important benefits of immutability: side-effect-free functions and data consistency.

Given that data can't be mutated, you can treat most functions as side-effect-free transformations. They take an input and return a result. More complicated programs are written by combining simpler transformations:

```
def complex_transformation(data) do
  data
  |> transformation_1(...)
  |> transformation_2(...)
  ...
  |> transformation_n(...)
end
```

This code relies on the previously mentioned pipeline operator that chains two functions together, feeding the result of the previous call as the first argument of the next call.

Side-effect-free functions are easier to analyze, understand, and test. They have well-defined inputs and outputs. When you call a function, you can be sure that no variable will be implicitly changed. Whatever the function does, you must take its result and do something with it.

**Note**

Elixir isn't a pure functional language, so functions may still have side effects. For example, a function may write something to a file and issue a database or network call, which causes it to produce a side effect. But you can be certain that a function won't modify the value of any variable.

The implicit consequence of immutable data is the ability to hold all versions of a data structure in the program. This in turn makes it possible to perform atomic in-memory operations. Let's say you have a function that performs a series of transformations:

```
def complex_transformation(original_data) do
```

```
  original_data
  |> transformation_1(...)
  |> transformation_2(...)
  ...
end
```

This code starts with the original data and passes it through a series of transformations, each one returning the new, modified version of the input. If something goes wrong, the function `complex_transformation` can return `original_data`, which will effectively roll back all of the transformations performed in the function. This is possible because none of the transformations modifies the memory occupied by `original_data`.

This concludes the basic immutability theory. It may still be unclear how to properly use immutable data in more complex programs. This topic will be revisited in chapter 4, where you deal with higher-level data structures.

### 2.4.6. Maps

A *map* is a key-value store, where keys and values can be any term. This is a new data type, introduced in Erlang/OTP 17.0 (dated April, 2014.). In its current version, it doesn't perform well with large number of elements, and for this you should currently use `HashDict` (presented later in this chapter). Maps are appropriate when you want to combine a couple of fields into a single structure. This use case somewhat overlaps that of tuples, but it provides the advantage of allowing you to access fields by name.

The following snippet demonstrates how to creates a map:

```
iex(1)> bob = %{:name => "Bob", :age => 25, :works_at => "Initech"}
%{age: 25, name: "Bob", works_at: "Initech"}
```

If keys are atoms, you can write this so it's slightly shorter:

```
iex(2)> bob = %{name: "Bob", age: 25, works_at: "Initech"}
%{age: 25, name: "Bob", works_at: "Initech"}
```

To retrieve a field, you can use the access operator `[]`:

```
iex(3)> bob[:works_at]
"Initech"

iex(4)> bob[:non_existent_field]
nil
```

Atom keys again receive special syntax treatment. The following snippet fetches a value stored under the `:age` key:

```
iex(5)> bob.age
25
```

But with this syntax, you'll get an error if you try to fetch the nonexistent field:

```
iex(6)> bob.non_existent_field
** (KeyError) key :non_existent_field not found
```

To change a field value, you can use the following syntax:

```
iex(7)> next_years_bob = %{bob | age: 26}
%{age: 26, name: "Bob", works_at: "Initech"}
```

This syntax can be used to change multiple attributes as well:

```
iex(8)> %{bob | age: 26, works_at: "Initrode"}
%{age: 26, name: "Bob", works_at: "Initrode"}
```

But you can only modify values that already exist in the map. This constraint makes it possible to optimize the map update. By knowing that the map structure doesn't change, the fields list can be reused between both versions. Consequently, you use less memory, and the update operation is faster.

Of course, there are situations where you want to insert a new key-value pair into the map. Map manipulations are available in the `Map` module (http://elixir-lang.org/docs/stable/elixir/Map.html). To insert a new key-value pair (or modify the existing one), you can use the `Map.put/3` function:

```
iex(9)> Map.put(bob, :salary, 50000)
%{age: 25, name: "Bob", salary: 50000, works_at: "Initech"}
```

In addition, a more general-purpose `Dict` module (http://elixir-lang.org/docs/stable/elixir/Dict.html) is provided. This module can be used to manipulate any abstract key-value structure such as a `HashDict` or a keyword list (both presented a bit later in this chapter).

For example, using `Dict.put/3`, you can also modify the map:

```
iex(10)> Dict.put(bob, :salary, 50000)
%{age: 25, name: "Bob", salary: 50000, works_at: "Initech"}
```

Using `Dict` has a benefit of shielding you from possible structural change. You can later switch to some other key-value structure without changing the code that manipulates it. On the flip side, `Dict` operations are somewhat slower than direct calls to the `Map` module. In general, though, for most of today's use cases, you won't need to use these modules directly. As mentioned, maps are currently best suited to manage small structures. In such scenarios, you can perform typical operations by using pure access and update syntax. You'll see how this works in chapter 4, where we deal with functional abstractions.

### 2.4.7. Binaries and bitstrings

A *binary* is a chunk of bytes. You can create binaries by enclosing the byte sequence between `<<` and `>>` operators. The following snippet creates a 3 byte binary:

```
iex(1)> <<1, 2, 3>>
<<1, 2, 3>>
```

Each number represents a value of the corresponding byte. If you provide a byte value bigger than 255, it's truncated to the byte size:

```
iex(2)> <<256>>
<<0>>

iex(3)> <<257>>
<<1>>

iex(4)> <<512>>
<<0>>
```

You can specify the size of each value and thus tell the compiler how many *bits* to use for that particular value:

```
iex(5)> <<257::16>>
<<1, 1>>
```

This expression places the number 257 into 16 bits of consecutive memory space. The output indicates that you use 2 bytes, both having a value of 1. This is due to the binary representation of 257, which in 16-bit form is written 00000001 00000001.

The size specifier is in bits and need not be a multiplier of 8. The following snippet creates a binary by combining two 4-bit values:

```
iex(6)> <<1::4, 15::4>>
<<31>>
```

The resulting value has 1 byte and is represented in the output using the normalized form 31 (0001 1111).

If the total size of all the values isn't a multiplier of 8, the binary is called a *bitstring*—a sequence of bits:

```
iex(7)> <<1::1, 0::1, 1::1>>
<<5::size(3)>>
```

You can also concatenate two binaries or bitstrings with the operator <>:

```
iex(8)> <<1, 2>> <> <<3, 4>>
<<1, 2, 3, 4>>
```

There is much more that can be done with binaries, but for the moment we'll put them aside. The most important thing you need to know about binaries is that they're consecutive sequences of bytes. Binaries play an important role in support for strings.

### 2.4.8. Strings

It may come as a surprise, but Elixir doesn't have a dedicated string type. Instead, strings are represented by using either a binary or a list type.

#### Binary strings

The most common way to use strings is to specify them with the familiar double-quotes syntax:

```
iex(1)> "This is a string"
"This is a string"
```

The result is printed as a string, but underneath, it's a binary—nothing more than a consecutive sequence of bytes.

Elixir provides support for embedded string expressions. You can use `#{}` to place an Elixir expression in a string constant. The expression is immediately evaluated, and its string representation is placed at the corresponding location in the string:

```
iex(2)> "Embedded expression: #{3 + 0.14}"
"Embedded expression: 3.14"
```

Classical `\` escaping works as you're used to:

```
iex(3)> "\r \n \" \\"
```

And strings don't have to finish on the same line:

```
iex(4)> "
        This is
        a multiline string
        "
```

Elixir provides another syntax for declaring strings, so-called *sigils*. In this approach, you enclose the string inside `~s()`:

```
iex(5)> ~s(This is also a string)
"This is also a string"
```

Sigils can be useful if you want to include quotes in a string:

```
iex(6)> ~s("Do... or do not. There is no try." -Master Yoda)
"\"Do... or do not. There is no try.\" -Master Yoda"
```

There is also an uppercase version `~S` that doesn't handle interpolation or escape characters (`\`):

```
iex(7)> ~S(Not interpolated #{3 + 0.14})
"Not interpolated \#{3 + 0.14}"

iex(8)> ~S(Not escaped \n)
"Not escaped \\n"
```

Finally, there is a special *heredocs* syntax, which supports better formatting for multiline strings. Heredocs strings start with a triple double-quote. The ending triple double-quote must be on its own line:

```
iex(9)> """
        Heredoc must end on its own line """
        """
"Heredoc must end on its own line \"\"\"\n"
```

Because strings are binaries, you can concatenate them with the `<>` operator:

```
iex(10)> "String" <> " " <> "concatenation"
"String concatenation"
```

Many helper functions are available for working with binary strings. Most of them reside in the `String` module (http://elixir-lang.org/docs/stable/elixir/String.html).

#### Character lists

The alternative way of representing strings is to use single-quote syntax:

```
iex(1)> 'ABC'
'ABC'
```

This creates a *character list*, which is essentially a list of integers in which each element represents a single character.

The previous result is exactly the same as if you manually construct

the list of integers:

```
iex(2)> [65, 66, 67]
'ABC'
```

As you can see, even the runtime doesn't distinguish between a list of integers and a character list. When a list consists of integers that represent printable characters, it's printed to the screen in the string form.

Just like with binary strings, there are syntax counterparts for various definitions of character lists:

```
iex(3)> 'Interpolation: #{3 + 0.14}'
'Interpolation: 3.14'

iex(4)> ~c(Character list sigil)
'Character list sigil'

iex(5)> ~C(Unescaped sigil #{3 + 0.14})
'Unescaped sigil \#{3 + 0.14}'

iex(6)> '''
        Heredoc
        '''
'Heredoc\n'
```

Character lists aren't compatible with binary strings. Most of the operations from the `String` module won't work with character lists. In general, you should prefer binary strings over character lists. Occasionally, some functions may work only with character lists. This mostly happens with pure Erlang libraries. In this case, you can convert a binary string to a character list version using the `String.to_char_list/1` function:

```
iex(7)> String.to_char_list("ABC")
'ABC'
```

To convert a character list to a binary string, you can use `List.to_string/1`.

In general, you should prefer binary strings as much as possible, using character lists only when some third-party library (most often written in pure Erlang) requires it.

### 2.4.9. First-class functions

In Elixir, a function is a first-class citizen, which means it can be assigned to a variable. Here, assigning a function to a variable doesn't mean calling the function and storing its result to a variable. Instead, the function definition itself is assigned, and you can use the variable to call the function.

Let's look at some examples. To create a function variable, you can use the `fn` construct:

```
iex(1)> square = fn(x) ->
          x * x
        end
```

The variable `square` now contains a function that computes the square of a number. Because the function isn't bound to a global name, it's also called an *anonymous function* or a *lambda*.

You can call this function by specifying the variable name followed by a dot (`.`) and the arguments:

```
iex(2)> square.(5)
25
```

**Note**

You may wonder why the dot operator is needed here. The motivation behind the dot operator is to make the code more explicit. When you encounter a `square.(5)` expression in the source code, you know an anonymous function is being invoked. In contrast, the expression `square(5)` is invoking a named function defined somewhere else in the module. Without the dot operator, you'd have to parse the surrounding code to understand whether you're calling a named or an anonymous function.

Because functions can be stored in a variable, they can be passed as arguments to other functions. This is often used to allow clients to parameterize generic logic. For example, the function `Enum.each/2` implements the generic iteration—it can iterate over anything *enumerable*, such as lists. The function `Enum.each/2` takes two

arguments: an enumerable and a one-arity lambda (an anonymous function that accepts one argument). It iterates through the enumerable and calls the lambda for each element. The clients provide the lambda to specify what they want to do with each element.

The following snippet uses `Enum.each` to print each value of a list to the screen:

```
iex(3)> print_element = fn(x) -> IO.puts(x) end        <-- Defines the lambda
iex(4)> Enum.each(
          [1, 2, 3],
          print_element          <-- Passes the lambda to Enum.each
        )
1                                 Output printed by
2                                 the lambda
3
:ok                    <-- Return value of Enum.each
```

Of course, you don't need a temp variable to pass the lambda to `Enum.each`:

```
iex(5)> Enum.each(
          [1, 2, 3],
          fn(x) -> IO.puts(x) end        <-- Passes the lambda directly
        )
1
2
3
```

Notice how the lambda just forwards all arguments to `IO.puts`, doing no other meaningful work. For such cases, Elixir makes it possible to directly reference the function and have a more compact lambda definition. So instead of writing `fn(x) -> IO.puts(x) end`, you can write `&IO.puts/1`.

The `&` operator, also known as the *capture* operator, takes the full function qualifier—a module name, a function name, and an arity—and turns that function into a lambda that can be assigned to a variable. You can use the capture operator to simplify the call to `Enum.each`:

```
iex(6)> Enum.each(
          [1, 2, 3],
          &IO.puts/1
        )
```

Passes the lambda that delegates to IO.puts

The capture operator can also be used to shorten the lambda definition, making it possible to omit explicit argument naming. For example, you can turn this definition

```
iex(7)> lambda = fn(x, y, z) -> x * y + z end
```

into a more compact form:

```
iex(8)> lambda = &(&1 * &2 + &3)
```

This snippet creates a three-arity lambda. Each argument is referred to via the `&n` placeholder, which identifies the *n*th argument of the function. You can call this lambda like any other:

```
iex(9)> lambda.(2, 3, 4)
10
```

The return value 10 amounts to 2 * 3 + 4, as specified in the lambda definition.

**Closures**

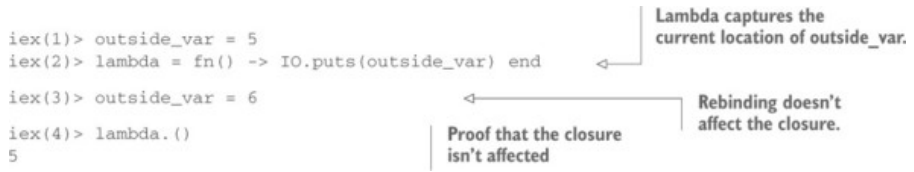
A lambda can reference any variable from the outside scope:

```
iex(1)> outside_var = 5
5
iex(2)> my_lambda = fn() ->
          IO.puts(outside_var)
        end
iex(3)> my_lambda.()
5
```

Lambda references a variable from the outside scope

As long as you hold the reference to `my_lambda`, the variable `outside_var` is also accessible. This is also known as *closure*: by holding a reference to a lambda, you indirectly hold a reference to all variables it uses, even if those variables are from the external scope.

A closure always captures a specific memory location. Rebinding a variable doesn't affect the previously defined lambda that references the same symbolic name:


iex(1)> outside_var = 5
iex(2)> lambda = fn() -> IO.puts(outside_var) end
iex(3)> outside_var = 6
iex(4)> lambda.()
5
Lambda captures the current location of outside_var.
Rebinding doesn't affect the closure.
Proof that the closure isn't affected

The previous code illustrates another important point. Normally, after you have rebound `outside_var` to the value 6, the original memory location would be eligible for garbage collection. But because the `lambda` function captures the original location (the one that holds the number 5), and you're still referencing that lambda, the original location isn't available for garbage collection.

### 2.4.10. Other built-in types

There are a couple of types I still haven't presented. We won't deal with them in depth, but it's worth mentioning them for the sake of completeness:

- *Reference* is an almost unique piece of information in a BEAM instance. It's generated by calling `Kernel.make_ref/0` (or `make_ref`). According to the Elixir documentation, a reference will reoccur after approximately 2^82 calls. But if you restart a BEAM instance, reference generation starts from the beginning, so its uniqueness is guaranteed only during the lifetime of the BEAM instance.
- A *pid* (process identifier) is used to identify an Erlang process. Pids are important when cooperating between concurrent tasks, and you'll learn about them in chapter 5 when we deal with Erlang processes.
- The *port identifier* is important when using ports. It's a mechanism used in Erlang to talk to the outside world. File I/O or communication with external programs are done through ports. Ports aren't the topic of this book.

With this, we've covered all the basic data types. As you can see, Elixir

has a simple type system consisting of only a handful of data types. Of course, higher-level types are also available, which build on these basic types to provide additional functionality; let's look at some of the most important ones that ship with Elixir.

### 2.4.11. Higher-level types

The built-in types just mentioned are inherited from the Erlang world. After all, Elixir code runs on BEAM, so its type system is heavily influenced by the Erlang foundations. But on top of these basic types, Elixir provides some higher-level abstractions. The ones most frequently used are `Range`, `Keyword`, `HashDict`, and `HashSet`. Let's examine each of them.

#### Range

A *range* is an abstraction that allows you to represent a range of numbers. Elixir even provides a special syntax for defining ranges:

```
iex(1)> range = 1..2
```

You can ask whether a number falls in the range by using the `in` operator:

```
iex(2)> 2 in range
true

iex(3)> -1 in range
false
```

Ranges are enumerable, so functions from the `Enum` module know how to work with them. Earlier, you met `Enum.each/2`, which iterates through an enumerable. The following example uses this function with a range to print the first three natural numbers:

```
iex(4)> Enum.each(
          1..3,                    <-------- Iterates over the range
          &IO.puts/1
        )

1
2
3
```

It's important to realize that a range isn't a special type. Internally, it's represented as a map which contains range boundaries. You shouldn't rely on this knowledge, because the range representation is an implementation detail. But it's good to be aware that the memory footprint of a range is very small, regardless of the size. A million-number range is still just a small map.

#### Keyword lists

A *keyword list* is a special case of a list, where each element is a two-element tuple, and the first element of each tuple is an atom. The second element can be of any type. Let's see an example:

```
iex(1)> days = [{:monday, 1}, {:tuesday, 2}, {:wednesday, 3}]
```

Elixir allows a slightly more elegant syntax for defining a keyword list:

```
iex(2)> days = [monday: 1, tuesday: 2, wednesday: 3]
```

Both constructs yield the same result: a list of pairs. Arguably, the second one is a bit more elegant.

Keyword lists are often used for small-size key-value structures, where keys are atoms. Many useful functions are available in the `Keyword` module (http://elixir-lang.org/docs/stable/elixir/Keyword.html). For example, you can use `Keyword.get/2` to fetch the value for a key:

```
iex(3)> Keyword.get(days, :monday)
1

iex(4)> Keyword.get(days, :noday)
nil
```

Just as with maps, you can use the access operator `[]` to fetch a value:

```
iex(5)> days[:tuesday]
2
```

Don't let that fool you, though. Because you're dealing with a list, the complexity of a lookup operation is O(n).

Keyword lists are most often useful to allow clients to pass an arbitrary number of optional arguments. For example, the result of the function `Float.to_string`, which converts a number to a binary string, can be controlled by providing additional options through a keyword list:

```
iex(6)> Float.to_string(1/3)
"3.33333333333333314830e-01"
```

Default behavior

```
iex(7)> Float.to_string(1/3, [decimals: 2])
"0.33"
```

Passes additional options

In fact, this pattern is so frequent that Elixir allows you to omit the square brackets if the last argument is a keyword list:

```
iex(8)> Float.to_string(1/3, decimals: 2, compact: true)
"0.33"
```

Notice in this example that you're still sending two arguments to `Float.to_string/2`: a number and a two-element keyword list. But this snippet demonstrates how to simulate optional arguments. You can accept a keyword list as the last argument of your function, and make that argument default to an empty list:

```
def my_fun(arg1, arg2, opts \\ []) do
  ...
end
```

Your clients can then pass options via the last argument. Of course, it's up to you to check the contents in the `opts` argument and perform some conditional logic depending on what the caller has sent you.

You may wonder if it's better to use maps instead of keywords for optional arguments. The main reason for using keyword lists is backward compatibility with existing Erlang and Elixir code. Maps are a recent addition to the Elixir/Erlang world, and before they were available, keywords were a standard approach to make functions accept various optional named arguments. There are some other special cases that keyword lists support. A keyword list can contain multiple values for the same key. In addition, you can control the ordering of keyword list elements—something that isn't possible with maps. Given that keyword lists perform fine if you keep them small, it's best to stick to the existing convention and accept optional parameters via keyword lists.

**HashDict**

A `HashDict` is a module that implements an arbitrarily sized key-value lookup structure. Unlike a map or a keyword list, a `HashDict` is more performant, especially for larger collections.

To create a `HashDict` instance, you can use `HashDict.new/0` function:

```
iex(1)> HashDict.new
#HashDict<[]>
```

To create a prepopulated `HashDict`, you can use `Enum.into/2`:

```
iex(2)> days = [monday: 1, tuesday: 2, wednesday: 3] |>
          Enum.into(HashDict.new)
#HashDict<[monday: 1, tuesday: 2, wednesday: 3]>
```

`Enum.into/2` is a generic function that can transfer anything enumerable into anything that is *collectable*. In this case, you transfer an enumerable keyword list into a collectable `HashDict`. More generally, you can transfer any enumerable whose elements are `{key, value}` pairs. A couple of examples are a list with `{key, value}` tuples (of which a keyword list is a special case), another `HashDict` instance, or a map.

**Note**

You might be confused by the concept of collectables. This is a complement to the enumerable concept. Whereas an enumerable is an abstract collection you can iterate on, a *collectable* is an abstract collection you can put elements into. Most provided collections, such as lists, maps, and `HashDict`s, are both enumerable and collectable. It's also possible to make your own data abstraction enumerable and/or collectable. We'll discuss this in detail in chapter 4.

As is the case with maps, functions from the `Dict` module can work with `HashDict` instances. But if you know you'll work directly with a `HashDict`, it's better to use the dedicated `HashDict` module.

To retrieve something from a dictionary, you can use the `HashDict.get/2` function:

```
iex(3)> HashDict.get(days, :monday)
1

iex(4)> HashDict.get(days, :noday)
nil
```

Similar to maps and keyword lists, the access `[]` operator can also be used:

```
iex(5)> days[:tuesday]
2
```

To modify a dictionary, you can use `HashDict.put/3`:

```
iex(6)> days = HashDict.put(days, :thursday, 4)

iex(7)> days[:thursday]
4
```

Many other operations can be done with dictionaries, and you're encouraged to research the `HashDict` module.

In addition, a `HashDict` instance is also an enumerable. This means you can use all the functions from the `Enum` module to iterate through the structure. When iterating, a lambda that you provide will receive a tuple in the form of a `{key, value}` pair for each element of the `HashDict`. Let's try this:

```
iex(8)> Enum.each(
          days,
          fn(key_value) ->
            key = elem(key_value, 0)
            value = elem(key_value, 1)
            IO.puts "#{key} => #{value}"
          end
        )

monday => 1
thursday => 4
tuesday => 2
wednesday => 3
```

As you can see, the ordering isn't guaranteed.

At the time of this writing, `HashDict` performs better than the map data type for large collections. Therefore, when you need a dynamically sized key-value structure, you should use `HashDict`. On the other hand, maps have a more elegant syntax, and they can be pattern-matched on (as you'll see in chapters 3 and 4). In time, the Erlang/OTP team may make maps more performant, so `HashDict` may end up being deprecated. Until then, it's best to limit `HashDict` usage to situations where it's needed (dynamic key-value stores) and use maps for all other cases (small, fix-sized structures).

#### HashSet

A `HashSet` is the implementation of a set. It works similarly to `HashDict`, with the exception that there are no pairs of data—it's a store of unique values, where a value can be of any type.

Here are some examples:

```
iex(1)> days = [:monday, :tuesday, :wednesday] |>
Enum.into(HashSet.new)
#HashSet<[:monday, :tuesday, :wednesday]>

iex(2)> HashSet.member?(days, :monday)
true

iex(3)> HashSet.member?(days, :noday)
false

iex(4)> days = HashSet.put(days, :thursday)
#HashSet<[:monday, :tuesday, :wednesday, :thursday]>
```

As you can see, this is all analogous to `HashDict`. Likewise, a `HashSet` is an enumerable:

```
iex(5)> Enum.each(days, &IO.puts/1)

monday
tuesday
wednesday
thursday
```

### 2.4.12. IO lists

An IO *list* is a special sort of list that is useful for incrementally building an output that will be forwarded to an I/O device such as

network or a file. Each element of an IO list must be one of the following:

1. An integer in the range of 0 to 255
2. A binary
3. An IO list

In other words, an IO list is a deep nested structure in which leaf elements are plain bytes (or binaries, which are again sequence of bytes). For example, here's `"Hello world"` represented as a convoluted IO list:

```
iex(1)> iolist = [[['H', 'e'], "llo,"], " worl", "d!"]
```

Notice how you can combine character lists and binary strings into a deep nested list. Many I/O functions can work directly and efficiently with such data. For example, you can print this structure to the screen:

```
iex(2)> IO.puts iolist
Hello, world!
```

Under the hood, the structure is flattened, and you can see the human-readable output. You'll get the same effect if you send an IO list to a file or a network socket.

IO lists are useful when you need to incrementally build a stream of bytes. Lists usually aren't good in this case, because appending to a list is an O(n) operation. In contrast, appending to an IO list is O(1), because you can use nesting. Here's a demonstration of this technique:

```
iex(3)> iolist = []                          <-- Initializes an IO list
        iolist = [iolist, "This"]
        iolist = [iolist, " is"]             |  Multiple appends
        iolist = [iolist, " an"]             |  to an IO list
        iolist = [iolist, " IO list."]

[[[[[], "This"], " is"], " an"], " IO list."]   <-- Final IO list
```

Here, you append to an IO list by creating a new list with two elements: a previous version of the IO list and the suffix that is appended. Each such operation is O(1), so this is performant. And of course, you can send this data to an IO function:

```
iex(4)> IO.puts iolist
This is an IO list.
```

This concludes our initial tour of the type system. We've covered most of the basics, and we'll expand on this theory throughout the book as the need arises. Next, it's time to learn a bit about Elixir operators.

## 2.5. Operators

We've been using various operators throughout this chapter, and in this section we'll take a systematic look at the ones most commonly used in Elixir. Most of the operators are defined in the `Kernel` module, and you can refer to the module documentation for a detailed description.

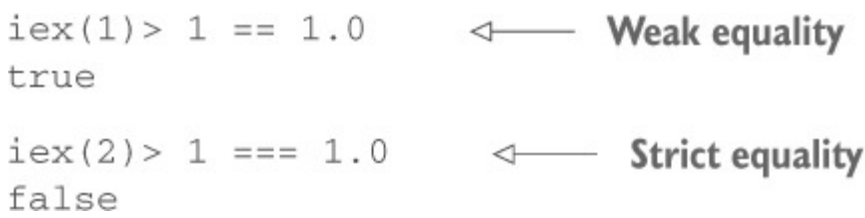
Let's start with arithmetic operators. These include the standard `+`, `-`, `*`, and `/`, and they work mostly as you'd expect, with the exception that the division operator always returns a float, as explained earlier in this chapter when we were dealing with numbers.

The comparison operators are more or less similar to what you're used to. They're listed in table 2.1.

**Table 2.1. Comparison operators**

| Operation | Description |
|---|---|
| `===`, `!==` | Strict equality/inequality |
| `==`, `!=` | Weak equality/inequality |
| `<`, `>`, `<=`, `>=` | Less than, greater than, less than or equal, greater than or equal |

The only thing we need to discuss here is the difference between strict and weak equality. This is relevant only when comparing integers to floats:




Logical operators work on boolean atoms. You saw this earlier, in the discussion of atoms, but let's repeat them once more: `and`, `or`, and `not`.

Unlike logical operators, short-circuit operators work with concept of *truthness*: the atoms `false` and `nil` are treated as falsy, and everything else is treated as truthy. The `&&` operator returns the second expression only if the first one isn't falsy. The `||` operator returns the first expression if it's truthy; otherwise it returns the second expression. The unary operator `!` returns false if the value is truthy; otherwise it returns true.

The operators presented here aren't the only ones available (for example, you've also seen the pipeline operator `|>`). But these are the most common ones, so it was worth mentioning them in one place.

**Many operators are functions**

It's worth noting that many operators in Elixir are actually functions. For example, instead of calling `a+b`, you can call `Kernel.+(a,b)`. Of course, no one would ever want to write this kind of code. But operator functions have a benefit when turned into anonymous functions. For example, you can create a two-arity lambda that sums two numbers by calling `&Kernel.+/2` or the shorter `&+/2`. Such lambdas can then be used with various enumeration and stream functions, as we'll explain in chapter 3.

We've almost completed our initial tour of the language. One thing remains: Elixir macros.

## 2.6. Macros

Macros are arguably one of the most important features Elixir brings to the table, compared to plain Erlang. They make it possible to perform powerful code transformations in compile time, thus reducing boilerplate and providing elegant, mini-DSL constructs.

Macros are a fairly complex subject, and it would take a small book to treat them extensively. Because this book is more oriented toward runtime and BEAM, and macros are a somewhat advanced feature that should be used sparingly, we won't provide a detailed treatment. But you should have a general idea of how macros work, because many Elixir features are powered by them.

A *macro* consists of Elixir code that can change the semantics of the input code. A macro is always called at compile time; it receives the parsed representation of the input Elixir code, and it has the opportunity to return an alternative version of that code.

Let's clear this up with an example. `unless` (an equivalent of `if not`) is a simple macro provided by Elixir:

```
unless some_expression do
  block_1
else
  block_2
end
```

`unless` isn't a special keyword. It's a macro (meaning an Elixir function) that transforms the input code into something like this:

```
if some_expression do
  block_2
else
  block_1
end
```

Such a transformation isn't possible with C-style macros, because the code of the expression can be arbitrarily complex and nested in multiple parentheses. But in Elixir macros (which are heavily inspired by LISP), you already work on a parsed source representation, so you'll have access to the expression and both blocks in separate

variables.

The end effect is that many parts of Elixir are written in Elixir with the help of macros. This includes the `unless` or `if` constructs, and also `defmodule` and `def`. Whereas other languages usually use keywords for such features, in Elixir they're built on top of a much smaller language core.

The main point to take away is that macros are compile-time code transformers. So whenever we say that something is a macro, the underlying implication is that it runs at compile time and produces alternative code.

**Special forms**

The Elixir compiler treats some language constructs in a special way. Such constructs are called *special forms* (http://elixir-lang.org/docs/stable/elixir/Kernel.SpecialForms.html). Some examples include the capture syntax `&(...)`, `for` comprehension (presented in chapter 3), the `receive` construct (chapter 5), and `try` blocks (chapter 8).

For details, you may want to look at the official meta-programming guide (http://elixir-lang.org/getting_started/meta/1.html). Meanwhile, we're done with our initial tour of the Elixir language. But before finishing this chapter, we should discuss some important aspects of the underlying runtime.

## 2.7. Understanding the runtime

As has been mentioned, the Elixir runtime is a BEAM instance. So once the compiling is done and the system is started, Erlang takes control. It's important to be familiar with some details of the virtual machine so you can understand how your systems work. First let's look at the significance of modules in the runtime.

### 2.7.1. Modules and functions in the runtime

Regardless of how you start the runtime, an OS process for the BEAM instance is started, and everything runs inside that process. This is true even when you're using the `iex` shell. If you need to find this OS process, you can look it up under the name `beam`.

Once the system is started, you run some code, typically by calling functions from modules. How does the runtime access the code? The VM keeps track of all modules loaded in memory. When you call a function from a module, BEAM first checks whether the module is loaded. If it is, then the code of the corresponding function is executed. Otherwise, the VM tries to find the compiled module file—the *bytecode*—on the disk and then load it and execute the function.

**Note**

The previous description reveals that each compiled module resides in a separate file. A compiled module file has the extension *.beam* (for Bogdan/Björn's Erlang Abstract Machine). The name of the file corresponds to the module name.

**Module names and atoms**

Let's recall how modules are defined:

```
defmodule Geometry do
  ...
end
```

Also recall from the discussion about atoms that `Geometry` is an alias that corresponds to `:"Elixir.Geometry"`, as demonstrated in the following snippet:

```
iex(1)> Geometry == :"Elixir.Geometry"
true
```

This isn't an accident. When you compile the source containing the `Geometry` module, the file generated on the disk is named `Elixir.Geometry.beam`, regardless of the name of the input source file. In fact, if multiple modules are defined in a single source file, compiler will produce multiple beam files that correspond to those modules. You can try this by calling the Elixir compiler (`elixirc`) from the command line

```
$ elixirc source.ex
```

where the file source.ex defines a couple of modules. Assuming there are no syntax errors, you'll see multiple beam files generated on the disk.

In the runtime, module names are aliases; and as we said, aliases are atoms. The first time you call the function of a module, BEAMtries to find the corresponding file on the disk. The VM looks for the file in the current folder and then in the *code paths*. When you start BEAM with Elixir tools (such as `iex`), some code paths are predefined for you. You can add additional code paths by providing the `-pa` switch:

```
$ iex -pa my/code/path -pa another/code/path
```

You can check which code paths are used at runtime by calling the Erlang function `:code.get_path`.

If the module is loaded, the runtime doesn't search for it on the disk. This can be used when starting the shell, to auto-load modules:

```
$ iex my_source.ex
```

This command compiles the source file and then immediately loads all generated modules. Notice that in this case, beam files aren't saved to disk. The `iex` tool performs an in-memory generation of bytecode generation and loads the modules.

Similarly, you can define modules in the shell:

**In-memory bytecode generation and loading of a module**

```
iex(1)> defmodule MyModule do
          def my_fun, do: :ok
        end

iex(2)> MyModule.my_fun
:ok
```

Again, the bytecode isn't saved to the disk in this case.

**Pure Erlang modules**

You've already seen how to call a function from a pure (non-Elixir) Erlang module. Let's talk a bit about this syntax:

```
:code.get_path
```

Calls the get_path function of the pure Erlang :code module

In Erlang, modules also correspond to atoms. Somewhere on the disk is a file named code.beam that contains the compiled code of the `:code` module. Erlang uses simple filenames, which is the reason for this call syntax. But the rules are the same as with Elixir modules. In fact, Elixir modules are nothing more than Erlang modules with fancier names (such as `Elixir.MyModule`).

You can create modules with simple names in Elixir (although this isn't recommended):

```
defmodule :my_module do
  ...
end
```

Compiling the source file that contains such a definition will generate my_module .beam on the disk.

The important thing to remember from this discussion is that at runtime, module names are atoms. And somewhere on the disk is an *xyz*.beam file, where *xyz* is the *expanded* form of an alias (such as `Elixir.MyModule` when the module is named `MyModule`).

#### Dynamically calling functions

Somewhat related to this discussion is the ability to dynamically call functions at runtime. This can be done with the help of the `Kernel.apply/3` function:

```
iex(1)> apply(IO, :puts, ["Dynamic function call."])
Dynamic function call.
```

`Kernel.apply/3` receives three arguments: the module atom, the function atom, and the list of arguments passed to the function. Together, these three arguments, often called MFA (for module, function, arguments), contain all the information needed to call an

exported (public) function. `Kernel.apply/3` can be useful when you need to make a runtime decision about which function to call.

### 2.7.2. Starting the runtime

There are multiple ways of starting BEAM. So far, we've been using `iex`, and we'll continue to do so for some time. But let's quickly look at all the possible ways to start the runtime.

#### Interactive shell

When you start the shell, the BEAM instance is started underneath, and the Elixir shell takes the control. The shell takes the input, *interprets* it, and prints the result.

It's important to be aware that input is interpreted, because it means it won't be as performant as the compiled code. This is generally fine, because you use the shell only to experiment with the language. But you shouldn't try to measure performance directly from `iex`.

On the other hand, modules are always compiled. Even if you define a module in the shell, it will be compiled and loaded in-memory, so there will be no performance hit.

#### Running scripts

The `elixir` command can be used to run a single Elixir source file. Here is the basic syntax:

```
$ elixir my_source.ex
```

When you start this, the following actions take place:

1. The BEAM instance is started.

2. The file my_source.ex is compiled in-memory, and the resulting modules are loaded to the VM. No beam file is generated

on the disk.

3. Whatever code resides outside of a module is *interpreted*.
4. Once everything is finished, BEAM is stopped.

This is most often useful for running scripts. In fact, it's recommended that such a script has a .exs extension, the trailing *s* indicating that it's a script.

This listing shows a simple Elixir script.

**Listing 2.5. Elixir script (script.exs)**

```
defmodule MyModule do
  def run do
    IO.puts("Called MyModule.run")
  end
end

MyModule.run
```

Code outside of a module is executed immediately.

You can execute this script from the command line:

```
$ elixir script.exs
```

This call first does the in-memory compilation of the `MyModule` module and then calls `MyModule.run`. After the call to `MyModule.run` finishes, the BEAM instance is stopped. If you don't want a BEAM instance to terminate, you can provide the `--no-halt` parameter:

```
$ elixir --no-halt script.exs
```

This is most often useful if your main code (outside a module) just starts concurrent tasks that perform all the work. In this case, your main call finishes as soon as these concurrent tasks are started, and BEAM is immediately terminated (and no work is done). Providing the `--no-halt` option keeps the entire system alive and running.

**The mix tool**

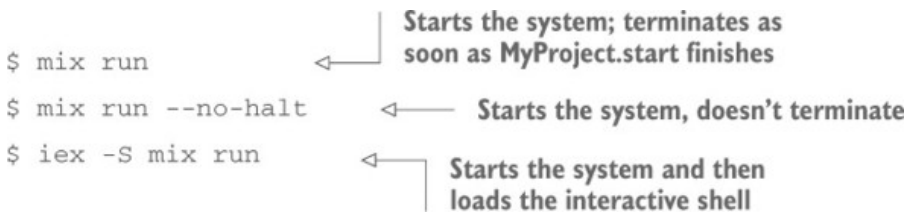
The `mix` tool is used to manage projects that are made up of multiple source files. Whenever you need to build a production-ready system, `mix` is your best option. To create a new mix project, you can call `mix new project_name` from the command line:

```
$ mix new my_project
```

This creates a new folder named my_project containing a couple of subfolders and files. You can change to the my_project folder and compile the entire project:

```
$ cd my_project
$ mix compile

Compiled lib/my_project.ex
Compiled lib/my_project/supervisor.ex
Generated my_project.app
```

The compilation goes through all the files from the lib and places the resulting beam files in the ebin folder. You can now start the mix project in multiple ways:


$ mix run
Starts the system; terminates as soon as MyProject.start finishes
$ mix run --no-halt
Starts the system, doesn't terminate
$ iex -S mix run
Starts the system and then loads the interactive shell

Regardless of how you start the mix project, it ensures that the ebin folder (where the beam files are placed) is in the load path so the VM can find your modules.

You'll use `mix` a lot, once you start creating more complex systems. For now, there's no need to go into any more detail.

## 2.8. Summary

This chapter introduced a lot of material. It's not a problem if you don't remember every detail, but you should be aware of the most

important points:

- Elixir code is divided into modules and functions.
- Elixir is a dynamic language. The type of a variable is determined by the value it holds.
- Data is immutable—it can't be modified. A function can return the modified version of the input that resides in another memory location. The modified version shares as much memory as possible with the original data.
- The most important primitive data types are numbers, atoms, and binaries.
- There is no boolean type. Instead, the atoms `true` and `false` are used.
- There is no nullability. The atom `nil` can be used for this purpose.
- There is no string type. Instead, you can use either binaries (recommended) or lists (when needed).
- The *only* complex types are tuples, lists, and maps. Tuples are used to group a small, *fixed-size* number of fields. Lists are used to manage *variable-size* collections. A map is a key-value data structure.
- `Range`, keyword lists, `HashDict`, and `HashSet` are abstractions built on top of the existing data system. They aren't distinct types.
- Functions are first-class citizens.
- Module names are atoms (or aliases) that correspond to *beam* files on the disk.
- There are multiple ways of starting programs: `iex`, `elixir`, and the `mix` tool.

This concludes your first contact with Elixir. Equipped with this knowledge, you can now start doing more exciting stuff. The first thing we'll do is take a look at some basic idioms of functional programming.

# Chapter 3. Control flow

*This chapter covers*

- Understanding pattern matching
- Working with multiclause functions
- Using conditional expressions
- Working with loops

Now that you're familiar with Elixir's basic building blocks, it's time to look at some typical low-level idioms of the language. In this chapter, we'll deal with conditionals and loops. As you'll see, these work differently than in most modern, imperative languages.

Classical conditional constructs such as `if` and `case` are often replaced with *multiclause functions*, and there are no classical loop statements such as `while`. But you can still solve problems of arbitrary complexity in Elixir, and the resulting code is no more complicated than a typical OO solution.

All this may sound a bit radical, which is why conditional and loops receive a detailed treatment in this chapter. But before we start discussing branching and looping, you need to learn about the important underlying supporting mechanism: pattern matching.

## 3.1. Pattern matching

As mentioned in chapter 2, the `=` operator isn't an assignment. Instead, when we wrote `a = 1`, we said variable `a` was *bound* to the value 1. The operator `=` is called the match operator, and the assignment-like expression is an example of *pattern matching*.

Pattern matching is an important construct in Elixir. It's a feature that makes manipulations with complex variables (such as tuples and lists) a lot easier. Less obviously, it allows you to write elegant, declarative-

like conditionals and loops. You'll see what this means by the end of the chapter; in this section, we'll look at the basic mechanical workings of pattern matching. Let's begin by looking at the match operator.

### 3.1.1. The match operator

So far, you have seen the most basic usage of the match operator:

```
iex(1)> person = {"Bob", 25}
```

We treated this as something akin to an assignment, but in reality something more complex is going on here. At runtime, the left side of the `=` operator is matched to the right side. The left side is called a *pattern*, whereas on the right side you have an expression that evaluates to an Elixir term.

In the example, you *match* the variable `person` to the right-side term `{"Bob", 25}`. A variable always matches the right-side term, and it becomes *bound* to the value of that term. This seems a bit theoretical, so let's look at a slightly more complex usage of the match operator that involves tuples.

### 3.1.2. Matching tuples

The following example demonstrates basic pattern matching of tuples:

```
iex(1)> {name, age} = {"Bob", 25}
```

This expression assumes that the right-side term is a tuple of two elements. When the expression is evaluated, the variables `name` and `age` are bound to the corresponding elements of the tuple. You can now verify that these variables are correctly bound:

```
iex(2)> name
"Bob"

iex(3)> age
25
```

This feature is useful when you call a function that returns a tuple and you want to bind individual elements of that tuple to separate variables. The following example calls the Erlang function `:calendar.local_time/0` to get the current date and time:

```
iex(4)> {date, time} = :calendar.local_time
```

The date and time are also tuples, which you can further decompose:

```
iex(5)> {year, month, day} = date

iex(6)> {hour, minute, second} = time
```

What happens if the right side doesn't correspond to the pattern? The match fails, and an error is raised:

```
iex(7)> {name, age} = "can't match"
** (MatchError) no match of right hand side value: "can't match"
```

**Note**

We didn't yet cover the error-handling mechanisms. This topic is treated in chapter 8. For now, suffice to say that raising an error works somewhat similarly to the classical exception mechanisms from mainstream languages. Therefore, when an error is raised, control is immediately transferred to code somewhere up the call chain, which catches the error (assuming such code exists).

Finally, it's worth noting that just like any other expression, the match expression also returns a value. The result of a match expression is always the right-side term you match against:

```
iex(8)> {name, age} = {"Bob", 25}
{"Bob", 25}
```

Match expression

Result of the match expression

Matching isn't confined to destructuring tuple elements to individual variables. Surprisingly enough, even constants are allowed on the left side of the match expression.

### 3.1.3. Matching constants

The left-side pattern can also include constants:

```
iex(1)> 1 = 1
1
```

Recall that the match operator `=` tries to match the right-side term to the left-side pattern. In the example, you try to match the pattern `1` to the term `1`. Obviously this succeeds, and the result of the entire expression is the right-side term.

The example doesn't have much practical benefit, but it illustrates that you can place constants to the left of `=`, which proves that `=` is not an assignment operator.

Constants are much more useful in compound matches. For example, it's common to use tuples to group various fields of a record. The following snippet creates a tuple that holds a person's name and age:

```
iex(2)> person = {:person, "Bob", 25}
```

The first element is a constant atom `:person`, which you use to denote that this tuple represents a person. Later you can rely on this knowledge and retrieve individual attributes of the person:

```
iex(3)> {:person, name, age} = person
{:person, "Bob", 25}
```

Here you expect the right-side term to be a three-element tuple, with its first element having a value of `:person`. After the match, the remaining elements of the tuple are bound to the variables `name` and `age`, which you can easily verify:

```
iex(4)> name
"Bob"

iex(5)> age
25
```

This is a common idiom in Elixir. Many functions from Elixir and Erlang return either `{:ok, result}` or `{:error, reason}`. For example, imagine that your system relies on a configuration file and expects it to always be available. You can read the file contents with

the help of the `File.read/2` function:

```
{:ok, contents} = File.read("my_app.config")
```

In this single line of code, three distinct things happen:

1. An attempt to open and read the file my_app.config takes place.
2. If the attempt succeeds, the file contents are extracted to the variable `contents`.
3. If the attempt fails, an error is raised. This happens because the result of `File.read` is a tuple in the form `{:error, reason}`, and therefore the match to `{:ok, contents}` fails.

By using constants in patterns, you tighten the match, making sure some part of the right side has a specific value.

### 3.1.4. Variables in patterns

Whenever a variable name exists in the left-side pattern, it always matches the corresponding right-side term. In addition, the variable is bound to the term it matches.

Occasionally we aren't interested in a value from the right-side term, but we still need to match on it. For example, let's say you want to get the current time of day. You can use the function `:calendar.local_time/0`, which returns a tuple `{date, time}`. But you aren't interested in a date, so you don't want to store it to a separate variable. In such cases, you can use the *anonymous variable* (_):

```
iex(1)> {_, time} = :calendar.local_time

iex(2)> time
{20, 44, 18}
```

When it comes to matching, the anonymous variable works just like a

named variable: it matches any right-side term. But the value of the term isn't bound to any variable.

Patterns can be arbitrarily nested. Taking the example further, let's say you only want to retrieve the current hour of the day:

```
iex(3)> {_, {hour, _, _}} = :calendar.local_time

iex(4)> hour
20
```

A variable can be referenced multiple times in the same pattern. In the following expressions, you expect an RGB triplet with the same number for each component:

```
iex(5)> {amount, amount, amount} = {127, 127, 127}
{127, 127, 127}

iex(6)> {amount, amount, amount} = {127, 127, 1}
** (MatchError) no match of right hand side value: {127, 127, 1}
```

Matches a tuple with three identical elements

Fails because the tuple elements aren't identical

Occasionally, you need to match against the contents of the variable. For this purpose, the *pin operator* (`^`) is provided. This is best explained with an example:

```
iex(7)> expected_name = "Bob"
"Bob"

iex(8)> {^expected_name, _} = {"Bob", 25}
{"Bob", 25}

iex(9)> {^expected_name, _} = {"Alice", 30}
** (MatchError) no match of right hand side value: {"Alice", 30}
```

Matches anything and then binds to the variable expected_name

Matches to the content of the variable

Matches to the content of the variable expected_name

Using `^expected_name` in patterns says that you expect the *value* of the variable `expected_name` to be in the appropriate position in the right-side term. In this example, it would be the same as if you used the hard-coded pattern `{"Bob", _}`. Therefore, the first match succeeds, but the second one fails.

Notice that the pin operator doesn't bind the variable. You expect that the variable is already bound—in other words, that it has a value—and you try to match against that value. This technique is used less often and is mostly relevant when you need to construct the pattern at

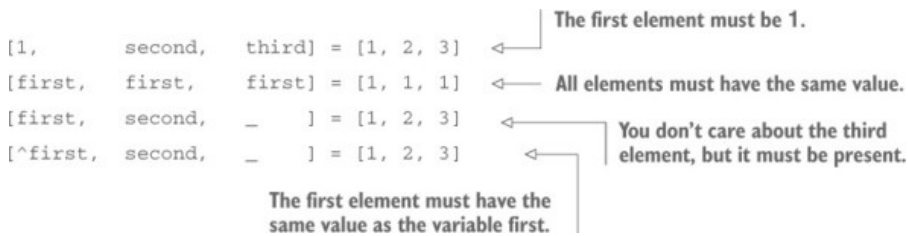
runtime.

### 3.1.5. Matching lists

List matching works similarly to tuples. The following example decomposes a three-element list:

```
iex(1)> [first, second, third] = [1, 2, 3]
[1, 2, 3]
```

And of course, the previously mentioned pattern techniques work as well:


[1, second, third] = [1, 2, 3]
The first element must be 1.
[first, first, first] = [1, 1, 1]
All elements must have the same value.
[first, second, _ ] = [1, 2, 3]
You don't care about the third element, but it must be present.
[^first, second, _ ] = [1, 2, 3]
The first element must have the same value as the variable first.

Matching lists is more often done by relying on their recursive nature. Recall from chapter 2 that each non-empty list is a recursive structure that can be expressed in the form `[head | tail]`. You can use pattern matching to put each of these two elements into separate variables:

```
iex(3)> [head | tail] = [1, 2, 3]
[1, 2, 3]

iex(4)> head
1

iex(5)> tail
[2, 3]
```

If you need only one element of the (head, tail) pair, you can use the anonymous variable. Here is an inefficient way of calculating the smallest element in the list:

```
iex(6)> [min | _] = Enum.sort([3,2,1])

iex(7)> min
1
```

First you sort the list, and then, with the pattern `[min | _]`, you take only the head of the (sorted) list. Note that this could also be done with the `hd` function mentioned in chapter 2. In fact, for this case, `hd` would be more elegant. The pattern `[head | _]` is more useful when pattern-matching function arguments, as you'll see in section 3.2.

### 3.1.6. Matching maps

To match a map, the following syntax can be used:

```
iex(1)> %{name: name, age: age} = %{name: "Bob", age: 25}
%{age: 25, name: "Bob"}

iex(2)> name
"Bob"

iex(3)> age
25
```

When matching a map, the left-side pattern doesn't need to contain all the keys from the right-side term:

```
iex(4)> %{age: age} =  %{name: "Bob", age: 25}

iex(5)> age
25
```

You may wonder about the purpose of such a partial-matching rule. Maps are frequently used to represent structured data. In such cases, you're often interested in only some of the map's fields. For example, in the previous snippet, you just want to extract the `age` field, ignoring everything else. The partial-matching rule allows you to do exactly this.

Of course, a match will fail if the pattern contains a key that's not in the matched term:

```
iex(6)> %{age: age, works_at: works_at} = %{name: "Bob", age: 25}
** (MatchError) no match of right hand side value
```

### 3.1.7. Matching bitstrings and binaries

We won't deal with bitstrings and pure binaries much in this book, but it's worth mentioning some basic matching syntax. Recall that a *bitstring* is a chunk of bits, and a *binary* is a special case of a bitstring that is always aligned to the byte size.

To match a binary, you use syntax similar to creating one:

```
iex(1)> binary = <<1, 2, 3>>
<<1, 2, 3>>

iex(2)> <<b1, b2, b3>> = binary
<<1, 2, 3>>

iex(3)> b1
1

iex(4)> b2
2

iex(5)> b3
3
```

A binary match

The example matches on a 3-byte binary and extracts individual bytes to separate variables.

The following example takes the binary apart by taking its first byte into one variable and the rest of the binary into another:

```
iex(6)> <<b1, rest :: binary>> = binary
<<1, 2, 3>>

iex(7)> b1
1

iex(8)> rest
<<2, 3>>
```

`rest::binary` states that you expect an arbitrary-sized binary. You can even extract separate bits or groups of bits. The following example splits a single byte into two 4-bit values:

```
iex(9)> <<a :: 4, b :: 4>> = <<155>>
<<155>>

iex(10)> a
9
```

```
iex(11)> b
11
```

Pattern `a::4` states that you expect a four-bit value. In the example, you put the first 4 bits into variable `a` and the other 4 bits into variable `b`. Because the number 155 is in binary represented as 10011011, you get values of 9 (1001 binary) and 11 (1011 binary).

Matching bitstrings and binaries is immensely useful when you're trying to parse packed binary content that comes from a file, an external device, or a network. In such situations, you can use binary matching to extract separate bits and bytes elegantly. As mentioned, the examples in the book won't need this feature. Still, you should make a mental note of binaries and pattern matching, in case the need arises at some point.

#### Matching binary strings

Recall that strings are binaries, so you can use binary matches to extract individual bits and bytes from the string:

```
iex(13)> <<b1, b2, b3>> = "ABC"
"ABC"

iex(13)> b1
65

iex(14)> b2
66

iex(15)> b3
67
```

Variables `b1`, `b2`, and `b3` hold corresponding bytes from the string you matched on. This isn't very useful, especially if you're dealing with unicode strings. Extracting individual characters is better done using functions from the `String` module.

A more useful pattern is to match the beginning of the string:

```
iex(16)> command = "ping www.example.com"
"ping www.example.com"

iex(17)> "ping " <> url = command      <-- Matching the string
"ping www.example.com"

iex(18)> url
"www.example.com"
```

In this example, you construct a string that holds a `ping` command. When you write `"ping " <> url = command`, you state the expectation that a `command` variable is a binary string starting with `"ping "`. If this matches, the rest of the string is bound to the variable `url`.

### 3.1.8. Compound matches

You've already seen this, but let's make it explicit. Patterns can be arbitrarily nested, as in the following contrived example:

```
iex(1)> [_, {name, _}, _] = [{"Bob", 25}, {"Alice", 30}, {"John",
35}]
```

In this example, the term being matched is a list of three elements. Each element is a tuple representing a person, consisting of two fields: the person's name and age. The match extracts the name of the second person in the list.

Another interesting feature is match chaining. Before you see how that works, let's discuss match expressions in more detail.

A match expression has this general form:

```
pattern = expression
```

As you've seen in examples, you can place any expression on the right side:

```
iex(2)> a = 1 + 3
4
```

Let's break down what happens here:

1. The *expression* on the right side is evaluated.
2. The resulting *value* is matched against the left-side pattern.
3. Variables from the pattern are bound.
4. The result of the match expression is the result of the right-side term.

An important consequence of this is that match expressions can be chained:

```
iex(3)> a = (b = 1 + 3)
4
```

In this (not so useful) example, the following things happen:

1. The expression `1 + 3` is evaluated.
2. The result (4) is matched against the pattern `b`.
3. The result of the inner match (which is again 4) is matched against the pattern `a`.

Consequently, both `a` and `b` have the value 4.

Parentheses are optional, and many developers omit them in this case:

```
iex(4)> a = b = 1 + 3
4
```

This yields the same result, due to the fact that the operator `=` is right-associative.

Now let's see a more useful example. Recall the function `:calendar.local_time/0`:

```
iex(5)> :calendar.local_time
{{2013, 11, 11}, {21, 28, 41}}
```

Let's say you want to retrieve the function's total result (datetime) as well as the current hour of the day. Here's the way to do it in a single compound match:

```
iex(6)> date_time = {_, {hour, _, _}} = :calendar.local_time
```

You can even swap the ordering. It still gives the same result (assuming you call it in the same second):

```
iex(7)> {_, {hour, _, _}} = date_time = :calendar.local_time
```

In any case, you get what you wanted:

```
iex(8)> date_time
{{2013, 11, 11}, {21, 32, 34}}

iex(9)> hour
21
```

This works because the result of a pattern match is always the result of the term being matched (whatever is on the right side of the match operator). Therefore, you can successively match against the result of that term and extract different parts you're interested in.

### 3.1.9. General behavior

We're almost done with basic pattern-matching mechanics. We've worked through a lot of examples, so let's try to formalize the behavior a bit.

The pattern-matching expression consists of two parts: the *pattern* (left side) and the *term* (right side). In a match expression, the attempt to match the term to the pattern takes place.

If the match succeeds, all variables in the pattern are bound to the corresponding values from the term. The result of the entire

expression is the entire term you matched. If the match fails, an error is raised.

Therefore, in a pattern-matching expression, you perform two different tasks:

1. You assert your expectations about the right-side term. If these expectations aren't met, an error is raised.
2. You bind some parts of the term to variables from the pattern.

The match operator `=` is just one example where pattern matching can be used. Pattern matching powers many other kinds of expressions, and it's especially powerful when used in functions.

## 3.2. Matching with functions

The pattern-matching mechanism is used in the specification of function arguments. Recall how the basic function definition looks:

```
def my_fun(arg1, arg2) do
  ...
end
```

The argument specifiers `arg1` and `arg2` are patterns, and you can use standard matching techniques.

Let's see this in action. As mentioned in chapter 2, tuples are often used to group related fields together. For example, if you do a geometry manipulation, you can represent a rectangle with a tuple `{a, b}` containing the rectangle's sides. The following listing shows a function that calculates a rectangle's area.

**Listing 3.1. Pattern-matching function arguments (rect.ex)**

```
defmodule Rectangle do
  def area({a, b}) do      <-- Matches a rectangle
    a * b
  end
end
```

Notice how you pattern-match the argument. The function `Rectangle.area/1` expects that its argument is a two-element tuple. It then binds corresponding tuple elements into variables and returns the result.

You can see whether this works from the shell. Start the shell, and load the module:

```
$ iex rect.ex
```

Then try the function:

```
iex(1)> Rectangle.area({2, 3})
6
```

What happens here? When you call a function, the arguments you provide are matched against the patterns specified in the function definition. The function expects a two-element tuple and binds the tuple's elements to variables `a` and `b`.

When calling functions, the term being matched is the argument provided to the function call. The pattern you match against is the argument specifier, in this case `{a, b}`.

Of course, if you provide anything that isn't a two-element tuple, an error will be raised:

```
iex(2)> Rectangle.area(2)
** (FunctionClauseError) no function clause matching in
Rectangle.area/1
     iex:2: Rectangle.area(2)
```

Pattern-matching function arguments is an extremely useful tool. It underpins one of the most important feature of Elixir: *multiclause*

*functions*.

### 3.2.1. Multiclause functions

Elixir allows you to *overload* a function by specifying multiple clauses. A *clause* is a function definition specified by the `def` construct. If you provide multiple definitions of the same function with the same arity, it's said that the function has multiple clauses.

Let's see this in action. Extending the previous example, let's say you need to develop a `Geometry` module that can handle various shapes. You'll represent shapes with tuples and use the first element of each tuple to indicate which shape it represents:

```
rectangle = {:rectangle, 4, 5}
square = {:square, 5}
circle = {:circle, 4}
```

Given these shape representations, you can write the following function to calculate a shape's area.

**Listing 3.2. Multiclause function (geometry.ex)**

```
defmodule Geometry do
  def area({:rectangle, a, b}) do      <-- First clause of area/1
    a * b
  end

  def area({:square, a}) do            <-- Second clause of area/1
    a * a
  end

  def area({:circle, r}) do            <-- Third clause of area/1
    r * r * 3.14
  end
end
```

As you can see, you provide three clauses of the same function. Depending on which argument you pass, the appropriate clause is called. Let's try this from the shell:

```
iex(1)> Geometry.area({:rectangle, 4, 5})
```

```
20

iex(2)> Geometry.area({:square, 5})
25

iex(3)> Geometry.area({:circle, 4})
50.24
```

When you call the function, the runtime goes through each of its clauses, in the order they're specified in the source code, and tries to match the provided arguments. The first clause that successfully matches all arguments is executed.

Of course, if no clause matches, an error is raised:

```
iex(4)> Geometry.area({:triangle, 1, 2, 3})
** (FunctionClauseError) no function clause matching in
Geometry.area/1
     geometry.ex:2: Geometry.area({:triangle, 1, 2, 3})
```

It's important to be aware that from the caller's perspective, a multiclause function is a single function. You can't directly reference a specific clause. Instead, you always work on the entire function. This applies to more than just function calls. Recall from chapter 2 that you can create a function value with the capture operator `&`:

```
&Module.fun/arity
```

If you capture `Geometry.area/1`, you capture all of its clauses:

```
iex(4)> fun = &Geometry.area/1      <-- Captures the entire function

iex(5)> fun.({:circle, 4})
50.24

iex(6)> fun.({:square, 5})
25
```

This proves that the function is treated as a whole, even if it consists of multiple clauses.

Sometimes you want a function to return a term indicating a failure, rather than raising an error. You can introduce a *default* clause that always matches. Let's do this for the area function. The next listing

adds a final clause that handles any invalid input.

**Listing 3.3. Multiclause function (geometry_invalid_input.ex)**

```
defmodule Geometry do
  def area({:rectangle, a, b}) do
    a * b
  end

  def area({:square, a}) do
    a * a
  end

  def area({:circle, r}) do
    r * r * 3.14
  end

  def area(unknown) do                          <-- Additional clause that handles invalid input
    {:error, {:unknown_shape, unknown}}
  end
end
```

If none of the first three clauses match, then the final clause is called. This is because a variable pattern always matches the corresponding term. In this case, you return a two-element tuple `{:error, reason}`, to indicate that something has gone wrong.

Try it from the shell:

```
iex(1)> Geometry.area({:square, 5})
25

iex(2)> Geometry.area({:triangle, 1, 2, 3})
{:error, {:unknown_shape, {:triangle, 1, 2, 3}}}
```

**Tip**

For this to work correctly, it's important to place the clauses in the appropriate order. The runtime tries to select the clauses using the order in the source code. If the `area(unknown)` clause was defined first, you would always get the error result.

Notice that the `area(unknown)` clause works only for `area/1`. If you pass more than one argument, this clause won't be called. Recall

from chapter 2 that functions differ in name and arity. Because functions with the same name but different arities are in reality two different functions, there is no way to specify an `area` clause that is executed regardless of how many arguments are passed.

One final note: you should always group clauses of the same function together, instead of scattering them around various places in the module. If a multiclause function is spread all over the file, it becomes increasingly hard to analyze the function's complete behavior. Even the compiler complains about this by emitting the compilation warning.

### 3.2.2. Guards

Let's say you want to write a function that accepts a number and returns an atom `:negative`, `:zero`, or `:positive`, depending on the number's value. This isn't possible with the simple pattern matching you have seen so far. Elixir gives you a solution for this in the form of *guards*.

Guards are an extension of the basic pattern-matching mechanism. They allow you to state additional broader expectations that must be satisfied for the entire pattern to match.

A guard can be specified by providing the `when` clause after the arguments list. This is best illustrated by example. The following code tests whether a given number is positive, negative, or zero.

**Listing 3.4. Using guards (test_num.ex)**

```
defmodule TestNum do
  def test(x) when x < 0 do
    :negative
  end

  def test(0), do: :zero

  def test(x) when x > 0 do
    :positive
  end
end
```

The guard is a logical expression that places further conditions on a clause. So the first clause will be called only if you pass a negative number, and the last one will be called only if you pass a positive number, as demonstrated in the shell session:

```
iex(1)> TestNum.test(-1)
:negative

iex(2)> TestNum.test(0)
:zero

iex(3)> TestNum.test(1)
:positive
```

Surprisingly enough, calling this function with a non-number yields strange results:

```
iex(4)> TestNum.test(:not_a_number)
:positive
```

What gives? The reason lies in the fact that Elixir terms can be compared with the operators `<` and `>`, even if they're not of the same type. In this case, the type ordering determines the result:

```
number < atom < reference < fun < port < pid <
  tuple < map < list < bitstring (binary)
```

A number is smaller than any other type, which is why `TestNum.test/1` always returns `:positive` if you provide a non-number. To fix this, you have to extend the guard by testing whether the argument is a number, as illustrated next.

**Listing 3.5. Using guards (test_num2.ex)**

```
defmodule TestNum do
  def test(x) when is_number(x) and x < 0 do
    :negative
  end

  def test(0), do: :zero

  def test(x) when is_number(x) and x > 0 do
    :positive
  end
end
```

This code uses the function `Kernel.is_number/1` to test whether the argument is a number. Now `TestNum.test/1` raises an error if

you pass a non-number:

```
iex(1)> TestNum.test(-1)
:negative

iex(2)> TestNum.test(:not_a_number)
** (FunctionClauseError) no function clause matching in
TestNum.test/1
```

The set of operators and functions that can be called from guards is very limited. In particular, you may not call your own functions, and most of the other functions won't work. The following operators and functions are allowed:

- Comparison operators (`==`, `!=`, `===`, `!==`, `>`, `<`, `<=`, `>=`)
- Boolean operators (`and`, `or`) and negation operators (`not`, `!`)
- Arithmetic operators (`+`, `-`, `*`, `/`)
- `<>` and `++` as long as the left side is a literal
- `in` operator
- Type-check functions from the `Kernel` module (for example, `is_number/1`, `is_atom/1`, and so on)
- Additional `Kernel` function `abs/1`, `bit_size/1`, `byte_size/1`, `div/2`, `elem/2`, `hd/1`, `length/1`, `map_size/1`, `node/0`, `node/1`, `rem/2`, `round/1`, `self/0`, `tl/1`, `trunc/1`, and `tuple_size/1`

Some of these functions may cause an error to be raised. For example, `length/1` makes sense only on lists. Imagine you have the following function that calculates the smallest element of a non-empty list:

```
defmodule ListHelper do
  def smallest(list) when length(list) > 0 do
    Enum.min(list)
  end
  def smallest(_), do: {:error, :invalid_argument}
end
```

You may think that calling `ListHelper.smallest/1` with anything other than list will raise an error, but this won't happen. If an error is raised from inside the guard, it won't be propagated, and the guard expression will return false. The corresponding clause won't match, but some other might.

In the example, if you call `ListHelper.smallest(123)`, you'll get the result `{:error, :invalid_argument}`. This demonstrates that an error in the guard expression is internally handled.

### 3.2.3. Multiclause lambdas

Anonymous functions (lambdas) may also consist of multiple clauses. First, let's recall the basic way of defining and using lambdas:

```
iex(1)> double = fn(x) -> x*2 end        <-- Defines a lambda
iex(2)> double.(3)                       <-- Calls a lambda
6
```

The general lambda syntax has the following shape:

```
fn
  pattern_1 ->
    ...                  <-- Executed if pattern_1 matches
  pattern_2 ->
    ...                  <-- Executed if pattern_2 matches
  ...
end
```

Let's see this in action by reimplementing the `test/1` function that inspects whether a number is positive, negative, or zero:

```
iex(3)> test_num = fn
          x when is_number(x) and x < 0 ->
            :negative
          0 -> :zero
          x when is_number(x) and x > 0 ->
            :positive
        end
```

Notice that there is no special ending terminator for a lambda clause. The clause ends when the new clause is started (in the form `pattern ->`) or when the lambda definition is finished with `end`.

You can now test this lambda:

```
iex(4)> test_num.(-1)
:negative

iex(5)> test_num.(0)
:zero

iex(6)> test_num.(1)
:positive
```

Multiclause lambdas come in handy when using higher-order functions, as you'll see later in this chapter. But for now, we're done with the basic theory behind multiclause functions. They play an important role in conditional runtime branching, which is our next topic.

## 3.3. Conditionals

Elixir provides some standard ways of doing conditional branching, with constructs such as `if` and `case`. Multiclause functions can be used for this purpose as well. In this section, we'll cover all the branching techniques, starting with multiclause functions.

### 3.3.1. Branching with multiclause functions

You've already seen how to do conditional logic with multiclauses, but let's repeat it once more:

```
defmodule TestNum do
  def test(x) when x < 0, do: :negative
  def test(0), do: :zero
  def test(x), do: :positive
end
```

The three clauses constitute three conditional branches. In a typical imperative language, such as JavaScript, you could write something like the following:

```
function test(x){
  if (x < 0) return "negative";
  if (x == 0) return "zero";
  return "positive";
}
```

Arguably, both versions are equally readable. The nice thing about multiclauses is that they can reap all the benefits of pattern matching. In the following example, a multiclause is used to test whether a given list is empty:

```
defmodule TestList do
  def empty?([]), do: true
  def empty?([_|_]), do: false
end
```

The first clause matches the empty list, whereas the second clause relies on the `[head | tail]` representation of a non-empty list.

Relying on pattern matching, you can implement polymorphic functions that do different things depending on the input type. The following example implements the function that doubles a variable. The function behaves differently depending on whether it's called with a number or with a binary (string):

```
iex(1)> defmodule Polymorphic do
          def double(x) when is_number(x), do: 2 * x

          def double(x) when is_binary(x), do: x <> x
        end

iex(2)> Polymorphic.double(3)
6

iex(3)> Polymorphic.double("Jar")
"JarJar"
```

The power of multiclauses starts to show in recursions. The resulting code seems declarative and is devoid of redundant `if`s and `return`s. Here's a recursive implementation of a factorial, based on multiclauses:

```
iex(4)> defmodule Fact do
          def fact(0), do: 1
          def fact(n), do: n * fact(n - 1)
        end

iex(5)> Fact.fact(1)
1

iex(6)> Fact.fact(3)
6
```

A multiclause-powered recursion is also used as a primary building block for looping. This is thoroughly explained in the next section, but here's a simple example. The following function sums all the elements of a list:

```
iex(7)> defmodule ListHelper do
          def sum([]), do: 0
          def sum([head | tail]), do: head + sum(tail)
        end

iex(8)> ListHelper.sum([])
0

iex(9)> ListHelper.sum([1, 2, 3])
6
```

The solution implements the sum by relying on the recursive definition of a list. The sum of an empty list is always zero, and a sum of a non-empty list equals the value of its head plus the sum of its tail.

The true elegance of multiclauses and pattern matches comes through when you have to combine functions that deal with different kinds of results. The following JavaScript snippet depicts a pattern you've probably seen many times:

```
var result = callSomeOperation(...);

if (result) {
  doSomething(result);
}
else {
  reportError();
}
```

This is a classical pattern where two functions need to be combined, the result of the first one being fed into the second one. But the first function may fail (returning `false` or `null`), and you need to check this with the corresponding `if` statement, which obfuscates the main flow of the code somewhat.

Let's see if you can do better in Elixir. You'll make a function that returns the number of lines in a file. For this, you'll use the `File.read/1` function, which returns either `{:ok, contents}` or `{:error, reason}` depending on whether the read operation succeeded. This is a frequent pattern in Elixir (and Erlang), and you'll

often encounter functions that return a result in such a form.

How can you handle different results of `File.read/1`? You'll implement a private function called `lines_num` that has separate clauses for success and failure cases. Here's the abstract sketch:

```
defp lines_num({:ok, contents}) do
  ...
end

defp lines_num(error) do
  ...
end
```

Given such a function, the "main" code can be absolved of conditional logic:

```
result = File.read(path)
lines_num(result)
```

Of course, you can get rid of the temporary `result` variable:

```
lines_num(File.read(path))
```

You can further remove the nested call by using the pipeline operator:

```
File.read(path)
|> lines_num
```

This code is self descriptive. You read the file and then count the lines of whatever you read. The conditional logic is pushed down to the multiclause `lines_num`, and the main code stays focused on the general workflow.

With that sketch in mind, let's look at the full implementation.

**Listing 3.6. Counting the lines in a file (lines_counter.ex)**

```
defmodule LinesCounter do
  def count(path) do
    File.read(path)
    |> lines_num
  end

  defp lines_num({:ok, contents}) do
    contents
    |> String.split("\n")
    |> length
  end

  defp lines_num(error), do: error
end
```

The "main" code

Success branch

Starts with contents

Splits it into a list of lines

Returns the length of the list

Error branch: returns the error

This code follows the presented idea, filling in the details we skipped in the first pass. The success branch handles the case when `File.read/1` manages to read a file. In this case, you can calculate the number of lines. To do this, you call `String.split/2`, which takes the string and returns the list of substrings, split by the given delimiter (newline in this case). The length of this list is the number of lines in the file.

The error branch handles any other case. Recall that a variable always matches, so this other clause is always executed, unless the first clause matched. If you arrive at the second clause, you can be sure that `File.read/1` failed, so you can handle the error somehow. In this case, you return the given error, effectively propagating it back to the caller. Alternatively, you could omit this second clause, which would cause the call to `lines_num/1` to fail (raise an error) for any input other than `{:ok, contents}`.

Let's see how this works:

```
iex(1)> LinesCounter.count("lines_counter.ex")
14

iex(2)> LinesCounter.count("non-existing file")
{:error, :enoent}
```

As you can see, the code works as expected, returning the number of lines in the first case or the corresponding error (generated from `File.read/1`) if the file could not be opened.

There is nothing you can do with multiclauses that can't be done with classical branching constructs. But the multiclause approach forces you to layer your code into many small functions and push the conditional logic deeper into lower layers. The underlying pattern-matching mechanism makes it possible to implement all kinds of branchings that are based on values and/or types of function arguments. The higher-level functions remain focused on the principal flow, and the entire code is arguably more self-descriptive.

In some cases, the code looks better with the classical, imperative style of branching. Let's look at what other branching constructs are in Elixir.

### 3.3.2. Classical branching constructs

Multiclause solutions may not always be appropriate. Using them requires creating a separate function and passing the necessary arguments. Sometimes it's simpler to use a classical branching construct in the function, and for such cases, the macros `if`, `unless`, `cond`, and `case` are provided. These work roughly as you might expect, although there are a couple of twists. Let's look at each one of these constructs.

#### If and unless

The `if` macro has a familiar syntax:

```
if condition do
  ...
else
  ...
end
```

This causes one or the other branch to execute, depending on the truthyness of the condition. If the condition is anything other than `false` or `nil`, you end up in the main branch; otherwise the `else` part is called.

You can also condense this into a one-liner, similarly to a `def` construct:

```
if condition, do: something, else: another_thing
```

Recall that everything in Elixir is an expression that has a return value. The `if` expression returns the result of the executed block (that is, of the block's last expression). If the condition isn't met and the `else` clause isn't specified, the return value is the atom `nil`:

```
iex(1)> if 5 > 3, do: :one
:one

iex(2)> if 5 < 3, do: :one
nil

iex(3)> if 5 < 3, do: :one, else: :two
:two
```

Let's see a more concrete example. The following code implements a `max` function that returns the larger of two elements (according to semantics of the `>` operator):

```
def max(a, b) do
  if a >= b, do: a, else: b
end
```

There is also the `unless` macro available, which is the equivalent of `if (not ...)`:

```
def max(a, b) do
  unless a >= b, do: b, else: a
end
```

**Cond**

The `cond` macro can be thought of as equivalent to an `if-else-if` pattern. It takes a list of expressions and executes the block of the first expression that evaluates to a truthy value:

```
cond do
  expression_1 ->
    ...
  expression_2 ->
    ...
  ...
end
```

The result of `cond` is the result of the corresponding executed block. If none of the conditions is satisfied, `cond` raises an error.

If you used `cond`, the simple `max/2` function could look like this:

```
def max(a, b) do
  cond do
    a >= b -> a
    true -> b      <-- Equivalent of a default clause
  end
end
```

This is fairly straightforward code, with the exception of the strange-looking `true -> b` part. The `true` pattern ensures that the condition will always be satisfied. If none of the previously stated conditions in the `cond` construct are met, the `true` branch is executed.

**Case**

The general syntax of `case` is as follows:

```
case expression do
  pattern_1 ->
    ...
  pattern_2 ->
    ...
  ...
end
```

The term *pattern* here indicates that it deals with pattern matching. In the `case` construct, the provided `expression` is evaluated, and then the result is matched against given clauses. The first one that matches is executed, and the result of the corresponding block (its last expression) is the result of the entire `case` expression. If no clause matches, an error is raised.

The `case`-powered version of the `max` function would then look like this:

```
def max(a,b) do
  case a >= b do
    true -> a
    false -> b
  end
end
```

The `case` construct is most suitable if you don't want to define a separate multiclause function. Other than that, there are no differences between `case` and multiclause functions. In fact, the general `case` syntax can be directly translated into the multiclause approach:

```
defp fun(pattern_1), do: ...
defp fun(pattern_2), do: ...
...
```

This must be called using the `fun(expression)`.

You can specify the default clause by using the anonymous variable to match anything:

```
case expression do
  pattern_1 -> ...
  pattern_2 -> ...
  ...
  _ -> ...      <—— The default clause that always matches
end
```

As you have seen, there are different ways of doing conditional logic in Elixir. Multiclauses offer a more declarative feel of branching, but they require defining a separate function and passing all the necessary arguments to it. Classical constructs like `if` or `case` seem more imperative but can often prove simpler than the multiclause approach. Selecting the appropriate solution depends on the specific situation as well as your personal preferences.

So far, we've covered basic Elixir branching techniques. Next it's time to look at how to do looping in Elixir.

## 3.4. Loops and iterations

Looping in Elixir works very differently than it does in mainstream languages. Constructs such as `while` and `do...while` aren't provided. If you think about it, they can't work, given that data is immutable.

Nevertheless, any serious program needs to do some kind of dynamic looping. So how do you go about it in Elixir? The principal looping tool is recursion. Therefore, we'll take a detailed look at how to use recursion for looping.

**Note**

Although recursion is the basic building block of any kind of looping, most production Elixir code uses it sparingly. The reason is that there are many higher-level abstractions that hide the recursion details. You'll learn about many of these abstractions throughout the book. Still, it's important to understand how recursion works in Elixir, because most of the complex code is based on this mechanism.

**Note**

Most of the examples in this section deal with simple problems, such as calculating the sum of all the elements in a list. Of course, Elixir gives us enough utility functions to do this in an effective and elegant one-liner. The point of the examples is to understand the different aspects of recursion-based processing on a simple problem.

### 3.4.1. Iterating with recursion

Let's say you want to implement a function that prints the first *n* natural numbers (positive integers). Because there are no loops, you must rely on recursion. The basic approach is illustrated in the following listing.

**Listing 3.7. Printing the first *n* natural numbers (natural_nums.ex)**

```
defmodule NaturalNums do
  def print(1), do: IO.puts(1)

  def print(n) do
    print(n - 1)
    IO.puts(n)
  end
end
```

This code relies on recursion, pattern matching, and multiclause functions. The code is very declarative: if `n` is equal to 1, you print the number. Otherwise, you print the first `n` – 1 numbers and then the nth one.

Trying it in the shell gives satisfying results:

```
iex(1)> NaturalNums.print(3)
1
2
3
```

You may have noticed that the function won't work correctly if you provide a negative integer or a float. This can be resolved with additional guards and is left for you as an exercise.

The code in listing 3.7 demonstrates the basic way of doing a conditional loop. You specify a multiclause function, first providing the clauses that stop the recursion. This is followed by more general clauses that produce part of the result and call the function recursively.

Next, let's look at how to compute something in a loop and return the result. You've already seen this example when dealing with conditionals, but let's repeat it. The following code implements a function that sums all the elements in a given list.

**Listing 3.8. Calculating the sum of the list (sum_list.ex)**

```
defmodule ListHelper do
  def sum([]), do: 0

  def sum([head | tail]) do
    head + sum(tail)
  end
end
```

This code looks very declarative:

- The sum of all the elements of an empty list is zero.
- The sum of all the elements of a non-empty list equals the list's head plus the sum of the list's tail.

Let's see it in action:

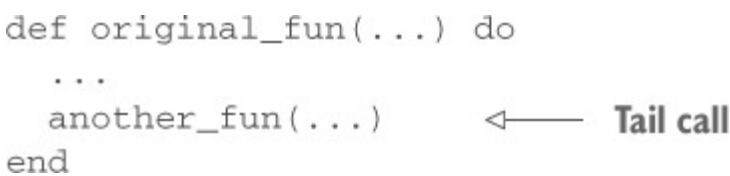
```
iex(1)> ListHelper.sum([1, 2, 3])
6

iex(2)> ListHelper.sum([])
```

0

The current implementation of the `sum/1` function has a potential problem. Given a large enough list, it might consume the entire memory! This is due to the fact that you call `sum` recursively: so for every element in the list, you allocate a bit of the memory, and you can ultimately run out. The solution to this problem lies in the so-called *tail-recursive* approach, which is based on *tail calls*, an important facility in functional programming languages.

### 3.4.2. Tail function calls

If the last thing a function does is call another function (or itself), you're dealing with a *tail call*:




Elixir (or, more precisely, Erlang) treats tail calls in a specific manner by performing a *tail-call optimization*. In this case, calling a function doesn't result in the usual stack push. Instead, something more like a goto or a jump statement happens. You don't allocate additional stack space before calling the function, which in turn means the tail function call consumes no additional memory.

How is this possible? In the previous snippet, the last thing done in `original_fun` is the call of `another_fun`. The final result of `original_fun` is the result of `another_fun`. This is why the compiler can safely perform the operation by jumping to the beginning of `another_fun` without doing additional memory allocation. When `another_fun` finishes, you return to whatever place `original_fun` was called from.

Tail calls are especially useful in recursive functions. A tail-recursive function—that is, a function that calls itself at the very end—can virtually run forever without consuming additional memory.

The following function is the Elixir equivalent of an endless loop:

```
def loop_forever(...) do
  ...
  loop_forever(...)
end
```

Because tail recursion doesn't consume additional memory, it's an appropriate solution for arbitrarily large iterations. There is a downside, though. Whereas classical (non-tail) recursion has a more declarative feel to it, tail recursion usually looks more procedural.

Let's convert the `ListHelper.sum/1` function to the tail-recursive version.

**Listing 3.9. Tail-recursive sum of the first *n* natural numbers (sum_list_tc.ex)**

```
defmodule ListHelper do
  def sum(list) do
    do_sum(0, list)
  end

  defp do_sum(current_sum, []) do
    current_sum
  end

  defp do_sum(current_sum, [head | tail]) do
    new_sum = head + current_sum
    do_sum(new_sum, tail)
  end
end
```

The first thing to notice is that you have two functions. The exported function `sum/1` is called by the module clients, and on the surface it works just like before.

The recursion takes place in the private `do_sum/2` function, which is implemented as tail recursive. It's a two-clause function, and we'll analyze it clause by clause. The second clause is more interesting, so we'll start with it. Let's repeat it here in isolation:

```
defp do_sum(current_sum, [head | tail]) do
  new_sum = head + current_sum
  do_sum(new_sum, tail)
end
```

Computes the new value of the sum

Tail-recursive call

This clause expects two arguments: the *non-empty list* to operate on,

and the sum you've calculated so far (`current_sum`). It then calculates the new sum and calls itself recursively with the remainder of the list and the new sum. Because the call happens at the very end, the function is tail recursive, and the call consumes no additional memory.

The variable `new_sum` is introduced here just to make things more obvious. You could also inline the computation:

```
defp do_sum(current_sum, [head | tail]) do
  do_sum(head + current_sum, tail)
end
```

And of course, you could use the pipeline operator:

```
defp do_sum(current_sum, [head | tail]) do
  head + current_sum
  |> do_sum(tail)
end
```

This function is still tail-recursive, because it calls itself at the very end.

The remaining thing to see is the first clause of `do_sum/2`:

```
defp do_sum(current_sum, []) do
  current_sum
end
```

This clause is responsible for stopping the recursion. It matches on an empty list, which is the last step of the iteration. When you get here, there is nothing else to sum, so you return the accumulated result.

Finally, you have the function `sum/1`:

```
def sum(list) do
  do_sum(0, list)
end
```

This function is used by clients and is also responsible for initializing the value of the `current_sum` parameter that is passed recursively in `do_sum`.

You can think of tail recursion as a direct equivalent of a classical loop in imperative languages. The parameter `current_sum` is a classical *accumulator*: the value where you incrementally add the result in each iteration step. The `do_sum/2` function implements the iteration step and passes the accumulator from one step to the next. Elixir is an immutable language, so you need this trick to maintain the accumulated value throughout the loop. The first clause of `do_sum/2` defines the ending point of the iteration and returns the accumulator value.

In any case, the tail-recursive version of the list sum is now working, and you can try it from the shell:

```
iex(1)> ListHelper.sum([1, 2, 3])
6

iex(2)> ListHelper.sum([])
0
```

As you can see, from the caller's point of view, the function works exactly the same. Internally, you rely on the tail recursion and can therefore process arbitrarily large lists without requiring extra memory for this task.

### Tail vs. non-tail recursion

Given the properties of tail recursion, you might think it's always a preferred approach of doing loops. It's not that simple. Non-tail recursion often looks more elegant and concise, and it can in some circumstances yield better performance. Therefore, when you write recursion, you should choose the solution that seems like a better fit. In cases when you expect possibly unlimited number of iterations, tail recursion is the only way. Otherwise, the choice amounts to what looks like a more elegant and performant solution.

#### Recognizing tail calls

Tail calls can take different shapes. You've seen the most obvious case, but there are a couple of others. A tail call can also happen in a conditional expression:

```
def fun(...) do
  ...
  if something do
    ...
    another_fun(...)      <---- Tail call
  end
end
```

The call to `another_fun` is a tail call because it's a last thing the function does. The same rule holds for `unless`, `cond`, and `case` expressions.

But the following code isn't a tail call:

```
def fun(...) do
  1 + another_fun(...)      <---- Not a tail call
end
```

This is because the call to `another_fun` isn't the last thing done in the `fun` function. After `another_fun` finishes, you have to increment its result by 1 to compute the final result of `fun`.

**Practicing**

All this may seem complicated, but it's not that hard. If you're coming from imperative languages, it's probably not what you're used to, and it takes some time to get accustomed to the recursive way of thinking, combined with the pattern-matching facility. You may want to take some time and experiment with recursion yourself. Here are a couple of functions you can write for practice:

- `list_len/1` function that calculates the length of a list
- `range/2` function that takes two integers: `from` and `to` and returns a list of all numbers in a given range
- `positive/1` function that takes a list and returns another list that contains only positive numbers from the input list

Try to write these functions first in the non-tail-recursive form, and

then convert them to the tail-recursive version. In case you get stuck, the solutions are provided in the recursion_practice.ex and recursion_practice_tc.ex files (for the tail-recursive versions).

Recursion is the basic looping technique, and no loop can be done without it. Still, you won't need to write explicit recursion all that often. Many typical tasks can be performed by using *higher-order functions*.

### 3.4.3. Higher-order functions

A higher-order function is a fancy name for a function that takes function(s) as its input and/or returns function(s). The word *function* here means *function value*. You've already made first contact with higher-order functions in chapter 2, when you used `Enum.each/2` to iterate through a list and print all of its elements. Let's recall how to do this:

```
iex(1)> Enum.each(
          [1, 2, 3],
          fn(x) -> IO.puts(x) end
        )

1
2
3
```

Passing a function value to another function

The function `Enum.each/2` takes an *enumerable* (in this case, a list, and a lambda. It iterates through the enumerable, calling the lambda for each of its elements. Because `Enum.each/2` takes a lambda as its input, it's called a higher-order function.

You can use `Enum.each/2` to iterate over enumerable structures without writing the recursion. Under the hood, `Enum.each/2` is of course powered by recursion: there is no other way to do loops and iterations in Elixir. But the complexity of writing the recursion, the repetitive code, and the intricacies of tail recursion are hidden from you.

`Enum.each/2` is just one example of an iteration powered by the

higher-order function. Elixir's standard library provides many other useful iteration helpers in the `Enum` module. The `Enum` module is a Swiss army knife for loops and iterations; it contains a lot of useful functions. You should spend some time researching the module documentation (http://elixir-lang.org/docs/stable/elixir/Enum.html). Here, we'll look at some of the most frequently used `Enum` functions.

### Enumerables

Most functions from the `Enum` module work on *enumerables*. You'll learn what this means in chapter 4. For now, it's sufficient to know that an enumerable is a data structure that implements a certain contract, which makes it suitable to be used by functions from the `Enum` module.

Some examples of enumerables are lists, ranges, `HashDict`, and `HashSet`. It's also possible to turn your own data structures into enumerables and thus harness all the features from the `Enum` module.

One manipulation you'll often need is a 1:1 transformation of a list to another list. For this purpose, `Enum.map/2` is provided. It takes an enumerable and a lambda that maps each element to another element. The following example doubles every element in the list:

```
iex(1)> Enum.map(
          [1, 2, 3],
          fn(x) -> 2 * x end
        )
[2, 4, 6]
```

Recall from chapter 2 that you can use the capture operator `&` to make the lambda definition a bit denser:

```
iex(2)> Enum.map(
          [1, 2, 3],
          &(2 * &1)
        )
```

The `&(...)` denotes a simplified lambda definition, where you use `&n` as placeholder for the *n*th argument of the lambda.

Another useful function is `Enum.filter/2`, which can be used to extract only some elements of the list, based on a certain criteria. The following snippet returns all odd numbers from a list:

```
iex(3)> Enum.filter(
          [1, 2, 3],
          fn(x) -> rem(x, 2) == 1 end
        )
[1, 3]
```

`Enum.filter/2` takes an enumerable and a lambda. It returns only those elements for which the lambda returns `true`.

Of course, you can use the capture syntax as well:

```
iex(3)> Enum.filter(
          [1, 2, 3],
          &(rem(&1, 2) == 1)
        )
[1, 3]
```

Probably the most versatile function from the `Enum` module is `Enum.reduce/3`, which can be used to transform an enumerable into anything. If you're coming from languages that supports first-class functions, you may already know `reduce` under a name *inject* or *fold*.

Reducing is best explained with a specific example. You'll use `reduce` to sum all the elements in a list. Before doing it in Elixir, let's see how you would do this task in an imperative manner. Here's an imperative JavaScript example:

```
var sum = 0;                          <-- Initializes the sum

[1, 2, 3].forEach(function(element) {
  sum += element;                     <-- Accumulates the result
})
```

This is a standard imperative pattern. You initialize an *accumulator* (variable `sum`) and then do some looping, adjusting the accumulator value in each step. After the loop is finished, the accumulator holds the final value.

In a functional language, you can't change the accumulator, but you

can still calculate the result incrementally by using `Enum.reduce/3`. The function has the following shape:

```
Enum.reduce(
  enumerable,
  initial_acc,
  fn(element, acc) ->
    ...
  end
)
```

`Enum.reduce/3` takes an enumerable as its first argument. The second argument is the initial value for the accumulator, the thing you compute incrementally. The final argument is a lambda that is called for each element. The lambda receives the current accumulator value and the element from the enumerable. The lambda's task is to compute and return the new accumulator value. When the iteration is done, `Enum.reduce/3` returns the final accumulator value.

Let's use `Enum.reduce/3` to sum up elements in the list:

```
iex(4)> Enum.reduce(
          [1, 2, 3],
          0,
          fn(element, sum) -> sum + element end
        )
6
```

Sets the initial accumulator value

Incrementally updates the accumulator

And that's all there is to it! Coming from an imperative background myself, it helps me to think of the lambda as the function that is called in each iteration step. Its task is to add a bit of the information to the result.

You may recall that I mentioned that many operators are functions, and you can turn an operator into a lambda by calling `&+/2`, `&*/2`, and so on. This combines nicely with higher-order functions. For example, the sum example can be written in a more compact form:

```
iex(5)> Enum.reduce([1,2,3], 0, &+/2)
6
```

It's of course worth mentioning that there is a function called `Enum.sum/1` that works exactly like this snippet. The point of the sum example was to illustrate how to iterate through a collection and

accumulate the result.

Let's work a bit more with `reduce`. The previous example works only if you pass a list that consists exclusively of numbers. If the list contains anything else, an error is raised (because the `+` operator is defined only for numbers). The next example can work on any type of list and sums only its numeric elements:

```
iex(6)> Enum.reduce(
          [1, "not a number", 2, :x, 3],
          0,
          fn                                          <-- Multiclause lambda
            element, sum when is_number(element) ->   <-- Matches numerical elements
              sum + element

            _, sum -> sum                             <-- Matches anything else
          end
        )
```

This example relies on a multiclause lambda to obtain the desired result. If the element is a number, you add its value to the accumulated sum. Otherwise (the element isn't a number), you return whatever sum you have at the moment, effectively passing it unchanged to the next iteration step.

Personally, I tend to avoid writing elaborate lambdas. If there is a bit more logic in the anonymous function, it's a sign that it will probably look better as a distinct function. In the following snippet, the lambda code is pushed to a separate private function:

```
defmodule NumHelper do
  def sum_nums(enumerable) do
    Enum.reduce(enumerable, 0, &add_num/2)    <-- Captures the add_num/2 to lambda
  end

  defp add_num(num, sum) when is_number(num), do: sum + num    <-- Handles each iteration step
  defp add_num(_, sum), do: sum
end
```

This is more or less similar to the approach you saw earlier. This example moves the iteration step to the separate, private function `add_num/2`. When calling `Enum.reduce`, you pass the lambda that delegates to that function, using the capture operator `&`.

Notice how, when capturing the function, you don't specify the module name. This is because `add_num/2` resides in the same

module, so you can omit the module prefix. In fact, because `add_num/2` is private, you can't capture it with the module prefix.

This concludes the basic showcase of the `Enum` module. Be sure to check the other functions that are available, because you'll find a lot of useful helpers that can simplify loops, iterations, and manipulations of enumerables.

### 3.4.4. Comprehensions

This cryptic name denotes another construct that can help you iterate and transform enumerables. The following example uses a comprehension to square each element of a list:

```
iex(1)> for x <- [1, 2, 3] do
          x*x
        end
```

The comprehension iterates through each element and runs the `do/end` block. The result is a list that contains all the results returned by the `do/end` block. In this basic form, `for` is no different than `Enum.map/2`.

Comprehensions have various other features that often makes them elegant, compared to `Enum`-based iterations. For example, it's possible to perform nested iterations over multiple collections. The following example takes advantage of this feature to calculate a small multiplication table:

```
iex(2)> for x <- [1, 2, 3], y <- [1, 2, 3], do: {x, y, x*y}

[
  {1, 1, 1}, {1, 2, 2}, {1, 3, 3},
  {2, 1, 2}, {2, 2, 4}, {2, 3, 6},
  {3, 1, 3}, {3, 2, 6}, {3, 3, 9}
]
```

In this example, the comprehension performs a nested iteration, calling the provided block for each combination of input collections.

Just like functions from the `Enum` module, comprehensions can iterate through anything that is enumerable. For example, you can use ranges to compute a multiplication table for single-digit numbers:

```
iex(3)> for x <- 1..9, y <- 1..9, do: {x, y, x*y}
```

In the examples so far, the result of the comprehension has been a list. But comprehensions can return anything that is collectable. *Collectable* is an abstract term for a functional data type that can collect values. Some examples include lists, maps, `HashDict`, and file streams; you can even make your own custom type collectable (more on that in chapter 4).

In more general terms, a comprehension iterates through enumerables, calling the provided block for each value and storing the results into some collectable structure. Let's see this in action. The following snippet makes a map that holds a multiplication table. Its keys are tuples of factors `{x,y}`, and the values contain products:

```
iex(4)> multiplication_table =
          for x <- 1..9, y <- 1..9,
              into: %{} do                    <-- Specifies the collectable
            {{x, y}, x*y}
          end

iex(5)> multiplication_table[{7, 6}]
42
```

Notice the `into` option, which specifies what to collect. In this case, it's an empty map `%{}` that will be populated with values returned from the `do` block. Notice how you return a `{factors, product}` tuple from the `do` block. You use this format because `map` "knows" how to interpret it. The first element will be used as a key, and the second one will be used as the corresponding value.

Another interesting comprehension feature is that you can specify filters. This gives you the possibility of skipping some elements from the input. The following example computes a nonsymmetrical multiplication table for numbers `x` and `y`, where `x` is never greater than `y`:

```
iex(6)> multiplication_table =
          for x <- 1..9, y <- 1..9,
              x <= y,                  <-- Comprehension filter
              into: %{} do
            {{x, y}, x*y}
          end

iex(7)> multiplication_table[{6, 7}]
42

iex(8)> multiplication_table[{7, 6}]
nil
```

The comprehension filter is evaluated for each element of the input enumerable, prior to block execution. If the filter returns true, then the block is called and the result is collected. Otherwise, the comprehension moves on to the next element.

As you can see, comprehensions are an interesting feature, allowing you to do some elegant transformations of the input enumerable. Although this can be done with `Enum` functions, most notably `Enum.reduce/3`, many times the resulting code looks more elegant when comprehensions are used. This is particularly true when you have to perform a Cartesian product (cross-join) of multiple enumerables, as was the case with the multiplication table.

**Note**

Comprehensions can also iterate through a binary. The syntax is somewhat different, and we won't treat it here. For details, it's best to look at the official `for` documentation at http://elixir-lang.org/docs/stable/elixir/Kernel.SpecialForms.html#for/1.

### 3.4.5. Streams

Streams are a special kind of enumerables that can be useful for doing lazy composable operations over anything enumerable. To see what this means, let's look at one shortcoming of standard `Enum` functions.

Let's say you have a list of employees, and you need to print each one prefixed by their position in the list:

```
1. Alice
2. Bob
3. John
...
```

This is fairly simple to perform by combining various `Enum` functions. For example, there is a function `Enum.with_index/1` that takes an enumerable and returns a list of tuples, where the first element of the tuple is a member from the input enumerable and the second element is its zero-based index:

```
iex(1)> employees = ["Alice", "Bob", "John"]
["Alice", "Bob", "John"]

iex(2)> Enum.with_index(employees)
[{"Alice", 0}, {"Bob", 1}, {"John", 2}]
```

You can now feed the result of `Enum.with_index/1` to `Enum.each/2` to get the desired output:

```
iex(3)> employees                                    |>
        Enum.with_index                              |>
        Enum.each(
          fn({employee, index}) ->
            IO.puts "#{index + 1}. #{employee}"
          end)
1. Alice
2. Bob
3. John
```

Here, you rely on the pipeline operator to chain together various function calls. This saves you from having to use intermediate variables and makes the code a bit cleaner.

**Pipeline operator in the shell**

You may wonder why the pipeline operator is placed at the end of the line. The reason is that in the shell, you have to place the pipeline on the same line as the preceding expression. Otherwise, as explained in chapter 2, the shell will immediately interpret the expression.

The pipeline operator at the end of the line signals to the shell that more input needs to be provided before the expression is complete. In the source file, it's better to place `|>` at the beginning of the next line.

So what's the problem with this code? Essentially, it iterates too much. The `Enum.with_index/1` function goes through the entire list to produce another list with tuples, and `Enum.each` then performs another iteration through the new list. Obviously, it would be better if you could do both operations in a single pass, and this is where streams can help you.

Streams are implemented in the `Stream` module (http://elixir-lang.org/docs/stable/elixir/Stream.html), which at first glance looks similar to the `Enum` module, containing functions like `map`, `filter`, and `take`. These functions take any enumerable as an input and give back a stream: an enumerable with some special powers.

A stream is a lazy enumerable, which means it produces the actual result on demand. Let's see what this means. The following snippet uses a stream to double each element in a list:

```
iex(4)> stream = [1, 2, 3] |>
          Stream.map(fn(x) -> 2 * x end)

#Stream<[enum: [1, 2, 3],
  funs: [#Function<44.45151713/1 in Stream.map/2>]]>
```

Creates the stream

The result of Stream.map/2 is a stream.

Because a stream is a lazy enumerable, the iteration over the input list (`[1, 2, 3]`) and the corresponding transformation (multiplication by 2) haven't yet happened. Instead, you get the structure that describes the computation.

To make the iteration happen, you have to send the stream to an `Enum` function, such as `each`, `map`, or `filter`. You can also use the `Enum.to_list/1` function, which converts any kind of enumerable into a list:

```
iex(5)> Enum.to_list(stream)
[2, 4, 6]
```

At this point, stream iteration takes place.

`Enum.to_list/1` (and any other `Enum` function, for that matter) is an eager operation. It immediately starts iterating through the input and creates the result. In doing so, `Enum.to_list/1` requests that the input enumerable start producing values. This is why the output of the stream is created when you send it to an `Enum` function.

The laziness of streams goes beyond iterating the list on demand. Values are produced one by one when `Enum.to_list` requests another element. For example, you can use `Enum.take/2` to request only one element from the stream:

```
iex(6)> Enum.take(stream, 1)
[2]
```

Because `Enum.take/2` iterates only until it collects the desired number of elements, the input stream doubled only one element in the list. The others were never visited.

Going back to the example of printing employees, using a stream allows you to print employees in a single go. The change to the original code is simple enough. Instead of using `Enum.with_index/1`, you can rely on its lazy equivalent, `Stream.with_index/1`:

```
iex(7)> employees                                          |>
        Stream.with_index                                  |>   <-- Performs a lazy transformation
        Enum.each(
          fn({employee, index}) ->
            IO.puts "#{index + 1}. #{employee}"
          end)

1. Alice
2. Bob
3. John
```

The output is the same, but the list iteration is done only once. This becomes increasingly useful when you need to compose multiple transformations of the same list. The following example takes the input list and prints the square root of only those elements that represent a non-negative number, adding an indexed prefix at the beginning:

```
iex(1)> [9, -1, "foo", 25, 49]                                    |>
        Stream.filter(&(is_number(&1) and &1 > 0))                 |>
        Stream.map(&{&1, :math.sqrt(&1)})                          |>
        Stream.with_index                                          |>
        Enum.each(
            fn({{input, result}, index}) ->
              IO.puts "#{index + 1}. sqrt(#{input}) = #{result}"
            end
          )

1. sqrt(9) = 3.0
2. sqrt(25) = 5.0
3. sqrt(49) = 7.0
```

This code is dense, and it represents how concise you can be by relying only on functions as the abstraction tool. You start with the input list and filter only positive numbers. You transform each such number into an `{input_number, square_root}` tuple. Then you index the resulting tuples using `Stream.with_index/1`, and, finally, you print the result.

Even though you stack multiple transformations, everything is performed in a single pass when you call `Enum.each`. In contrast, if you used `Enum` functions everywhere, you would have to run multiple iterations over each intermediate list, which would incur a performance and memory-usage penalty.

This lazy property of streams can become useful for consuming slow and potentially large enumerable input. A typical case is when you need to parse each line of a file. Relying on eager `Enum` functions means you have to read the entire file into memory and then iterate through each line. This is exactly what you did earlier, when counting the lines in a file in listing 3.6. In that example, you read the file's entire contents into memory, split it with newline characters, and finally calculated the length of the resulting list.

In contrast, using streams makes it possible to read and immediately parse one line at a time. For example, the following function takes a filename and returns the list of all lines from that file that are longer than 80 characters:

```
def large_lines!(path) do
  File.stream!(path)
  |> Stream.map(&String.replace(&1, "\n", ""))
  |> Enum.filter(&(String.length(&1) > 80))
end
```

Here you rely on the `File.stream!/1` function, which takes a path of the file and returns a stream of its lines. Because the result is a stream, the iteration through the file happens only when you request it. So after `File.stream!` returns, no byte from the file has been read yet. Then you remove the trailing newline character from each line, again in the lazy manner.

Finally, you eagerly take only long lines, using `Enum.filter/2`. It's

at this point that iteration happens. The consequence is that you never read the entire file in memory; instead, you work on each line individually.

#### Note

There are no special tricks in the Elixir compiler that allow these lazy enumerations. The real implementation is fairly involved, but the basic idea behind streams is simple and relies on anonymous functions. In a nutshell, to make a lazy computation, you need to return a lambda that performs the computation. This makes the computation lazy, because you return its description rather than its result. When the computation needs to be materialized, the consumer code can call the lambda.

#### Practicing

Of course, this style of coding takes some getting used to. You'll use the techniques presented here throughout the book, but you should try to write a couple such iterations yourself. Here are some exercise ideas that may help you get into the swing of things.

Using `large_lines!/1` as a model, write the following function:

- `lines_lengths!/1` that takes a file path and returns a list of numbers, with each number representing the length of the corresponding line from the file.
- `longest_line_length!/1` that returns the length of the longest line in a file.
- `longest_line!/1` that returns the contents of the longest line in a file.
- `words_per_line!/1` that returns a list of numbers, with each number representing the word count in a file. Hint: to get the word count of a line, use `length(String.split(line))`.

Solutions are provided in the enum_streams_practice.ex file, but it's strongly suggested that you spend some time trying to crack these problems yourself.

## 3.5. Summary

This has been a detailed treatment of conditional and branching constructs in Elixir. As you've seen, these work significantly differently than what you may have been used to. We covered a lot of theory in the process, and you've seen many examples, so let's recap the most important points:

- Pattern matching is a construct that attempts to match a right-side term to the left-side pattern. In the process, variables from the pattern are bound to corresponding subterms from the term. If a term doesn't match the pattern, an error is raised.
- Function arguments are patterns. Calling a function tries to match the provided values to the patterns specified in the function definition.
- Functions can have multiple clauses. The first clause that matches all the arguments is executed.
- Multiclause functions are a primary tool for conditional branching, with each branch written as a separate clause. Less often, classical constructs such as `if`, `unless`, `cond`, and `case` are used.
- Recursion is the main tool for implementing loops. Tail recursion is used when you need to run an arbitrarily long loop.
- Higher-order functions make writing loops much easier. There are many useful generic iteration functions in the `Enum` module. The `Stream` module additionally makes it possible to implement lazy and composable iterations.
- Comprehensions can also be used to iterate, transform, filter, and join various enumerables.

We're finished with the basics of functional programming techniques and idioms. Our exploration of functional programming continues in the next chapter, where you'll learn how to work with higher-level abstractions.

# Chapter 4. Data abstractions

*This chapter covers*

- Abstracting with modules
- Working with hierarchical data
- Polymorphism with protocols

This chapter deals with building higher-level data structures. In any complex system, there will be a need for abstractions such as `Money`, `Date`, `Employee`, and `OrderItem`, all textbook examples of higher-level abstractions that usually aren't directly supported by the language and are instead written on top of built-in types.

In Elixir, such abstractions are implemented with pure, stateless modules. In this chapter, you'll learn how to create and work with your own data abstractions.

In a typical OO language, the basic abstraction building blocks are classes and objects. For example, there may be a `String` class available that implements various string operations. Each string is then an instance of that class that can be manipulated by calling methods, as the following Ruby snippet illustrates:

```
"a string".upcase
```

This approach generally isn't used in Elixir. Being a functional language, Elixir promotes decoupling of data from the code. Instead of classes, you use *modules*, which are collection of functions. Instead of calling methods on objects, you explicitly call module functions and provide input data via arguments. The following snippet shows the Elixir way of uppercasing the string:

```
String.upcase("a string")
```

Another big difference from OO languages is that data is immutable. To modify data, you must call some function and take its result into a variable; the original data is left intact. The following examples

demonstrates this technique:

```
iex(1)> list = []
[]

iex(2)> list = List.insert_at(list, -1, :a)
[:a]

iex(3)> list = List.insert_at(list, -1, :b)
[:a, :b]

iex(4)> list = List.insert_at(list, -1, :c)
[:a, :b, :c]
```

In these examples, you're constantly keeping the result of the last operation and feeding it to the next one.

The important thing to notice in both Elixir snippets is that the module is used as the abstraction over data type. When you need to work with strings, you reach for the `String` module. When you need to work with lists, you use the `List` module.

`String` and `List` are examples of modules that are dedicated to a specific data type. They're implemented in pure Elixir, and their functions rely on the predefined format of the input data. `String` functions expect a binary string as the first argument, whereas `List` functions expect a list.

Additionally, *modifier functions* (the ones that transform the data) return data of the same type. The function `String.upcase/1` returns a binary string, whereas, for example `List.insert_at/3`, returns a list.

Finally, a module also contains *query functions* that return some piece of information from the data: for example, `String.length/1` and `List.first/1`. Such functions still expect an instance of the data abstraction as the first argument, but they return another type of information.

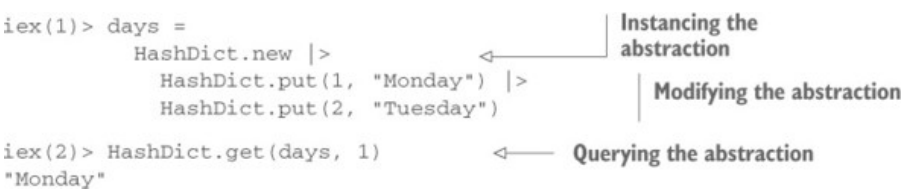
The basic principles of data abstraction in Elixir can be summarized as follows:

- A module is in charge of abstracting some data.
- The module's functions usually expect an instance of the data abstraction as the first argument.
- Modifier functions return a modified version of the abstraction.
- Query functions return some other type of data.

Given these principles, it's fairly straightforward to create your own higher-level abstractions, as you'll see in the next section.

## 4.1. Abstracting with modules

Lists and strings are admittedly lower-level types. But writing higher-level data abstractions is based on the same principles just stated. In fact, you've already seen examples of a higher-level data abstraction in chapter 2. For example, a `HashDict` module implements a key-value dictionary. `HashDict` is implemented in pure Elixir and can serve as a good template for how to design an abstraction in Elixir. Let's see an example of `HashDict` usage:




This approach more or less follows the principles stated earlier. The code is slightly simplified using the pipeline operator to chain operations together. This is possible due to the so-called "subject as first argument convention": functions of a module that is responsible for a data abstraction should receive that abstraction as the first argument. Such functions are pipeline friendly and can be chained with the `|>` operator.

Notice the `new/0` function that creates an empty instance of the abstraction. There is nothing special about this function, and it could have been given any name. Its only purpose is to create an empty data structure you can then work on.

Because `HashDict` is an abstraction, as a client of this module, you don't concern yourself with its internal working or its data structure. You call `HashDict` functions, holding on to whatever result you get and passing that result back to functions from the same module.

**Note**

You may think that abstractions like `HashDict` are something like user-defined types. Although there are many similarities, module-based abstractions aren't proper data types like the ones explained in chapter 2. Instead, they're implemented by composing built-in data types. For example, a `HashDict` instance is a map, which you can verify by invoking `is_map(HashDict.new)`.

Given this template, let's try to build a simple data abstraction.

### 4.1.1. Basic abstraction

The example in this section is a simple to-do list. The problem is admittedly not spectacular, but it's complex enough to give you something to work on while not being overly complicated. Thus we can focus on techniques without spending too much time trying to grasp the problem itself.

The basic version of the to-do list will support the following features:

- Creating a new data abstraction
- Adding new entries
- Querying the abstraction

Here's an example of the desired usage:

```
$ iex simple_todo.ex

iex(1)> todo_list =
          TodoList.new |>
            TodoList.add_entry({2013, 12, 19}, "Dentist") |>
            TodoList.add_entry({2013, 12, 20}, "Shopping") |>
            TodoList.add_entry({2013, 12, 19}, "Movies")

iex(2)> TodoList.entries(todo_list, {2013, 12, 19})
```

```
["Movies", "Dentist"]

iex(3)> TodoList.entries(todo_list, {2013, 12, 18})
[]
```

This is fairly self-explanatory. You instantiate a structure by calling `TodoList.new/0`, then add some entries, and finally execute some queries. The tuple `{2013, 12, 19}` represents a date in the `{year, month, day}` format.

As the chapter progresses, you'll add additional features and modify the interface slightly. You'll continue adding features throughout the book, and by the end you'll have a fully working distributed web server that can manage a large number of to-do lists.

But for now, let's start with this simple interface. First you have to decide on the internal data representation. Looking at the snippet, you can see that the primary use case is finding all entries for a single date. Therefore, using `HashDict` seems like a reasonable initial approach. You'll use dates as keys, with values being lists of entries for given dates. With this in mind, the implementation of the `new/0` function is straightforward.

**Listing 4.1. Initializing a to-do list (simple_todo.ex)**

```
defmodule TodoList do
  def new, do: HashDict.new
  ...
end
```

Next you have to implement the `add_entry/3` function. This function expects a to-do list (which you know is a `HashDict`) and has to add the entry to the list under a given key (date). Of course, it's possible that no entries for that date exist, so you have to cover that case as well. As it turns out, this can be done with a single call to the function `HashDict.update/4`.

**Listing 4.2. Adding an entry (simple_todo.ex)**

```
defmodule TodoList do
  ...

  def add_entry(todo_list, date, title) do
    HashDict.update(
      todo_list,
      date,
      [title],                                  <-- 1 Initial value
      fn(titles) -> [title | titles] end        <-- 2 Updater lambda
    )
  end

  ...
end
```

The `HashDict.update/4` function receives a dictionary, a key, an initial value (1), and an *updater lambda* (2). If no value exists for the given key, the initial value is used. Otherwise, the updater lambda is called. The lambda receives the existing value and returns the new value for that key. In this case, you push the new entry to the top of list. You may remember from chapter 2 that lists are most efficient when pushing new elements to the top. Therefore, you opt for a fast insertion operation but sacrifice ordering—more recently added entries are placed before the older ones in the list.

Finally, you need to implement the `entries/2` function that returns all entries for a given date or an empty list if no task exists for that date. This is fairly straightforward, as you can see in the next listing.

**Listing 4.3. Querying the to-do list (simple_todo.ex)**

```
defmodule TodoList do
  ...

  def entries(todo_list, date) do
    HashDict.get(todo_list, date, [])
  end
end
```

You fetch a value for the given date from `todo_list`, which must be a `HashDict` instance. The third argument to `HashDict.get/3` is a default value that is returned if a given key isn't present in the dictionary.

### 4.1.2. Composing abstractions

Of course, nothing stops you from creating one abstraction on top of another. In fact, `TodoList` is built on top of `HashDict`, which is itself a higher-level abstraction built on top of core data types.

Looking at the initial take on the to-do list, there is an opportunity to move some of the code into a separate abstraction. Look at the way you operate on a dictionary, allowing multiple values to be stored under a single key and retrieving all values for that key. This code could be moved to a separate abstraction. Let's call this `MultiDict`, which is implemented in the next listing.

**Listing 4.4. Implementing the MultiDict abstraction (todo–multi_dict.ex)**

```
defmodule MultiDict do
  def new, do: HashDict.new

  def add(dict, key, value) do
    HashDict.update(
      dict,
      key,
      [value],
      &[value | &1]
    )
  end

  def get(dict, key) do
    HashDict.get(dict, key, [])
  end
end
```

This is more or less a copy-and-paste from the initial to-do list implementation. The names are changed a bit, and you use the capture operator to shorten the updater lambda definition `&[value | &1]`.

With this abstraction in place, the `TodoList` module becomes much simpler.

**Listing 4.5. `TodoList` relying on a `MultiDict` (todo_multi_dict.ex)**

```
defmodule TodoList do
  def new, do: MultiDict.new

  def add_entry(todo_list, date, title) do
    MultiDict.add(todo_list, date, title)
  end

  def entries(todo_list, date) do
    MultiDict.get(todo_list, date)
  end
```

```
end
```

This is a classical separation of concerns, where you extract a distinct responsibility into a separate abstraction and then create another abstraction on top of it. A distinct `MultiDict` abstraction is now readily available to be used in other places in code, if needed. Furthermore, you can extend `TodoList` with additional functions that don't belong to `MultiDict`: for example, `due_today/2` would return all entries for today.

The point of this refactoring is to illustrate that the code organization isn't that different from an OO approach. Of course, you use different tools to create abstractions (stateless modules and pure functions instead of classes and methods), but the general idea is still the same.

### 4.1.3. Structuring data with maps

`TodoList` now works as a reasonable abstraction. You can insert entries into the structure and get all entries for a given date. But the interface is somewhat clumsy. When adding a new entry, you have to specify each field as a separate argument:

```
TodoList.add_entry(todo_list, {2013, 12, 19}, "Dentist")
```

If you want to extend an entry with another attribute—for example, time—you must change the signature of the function, which will in turn break all the clients. Moreover, you have to change every place in the implementation where this data is being propagated.

An obvious solution to this problem is to somehow combine all entry fields as a single data abstraction. In Elixir, the basic way of keeping different fields together is to include them in a tuple. For example, you're already doing this for the date field, where individual date parts are combined into a single `{year, month, day}` triplet.

Tuples can work well for some cases, but they have a problem: fields aren't named. You have to manually keep track of a tuple's size and the position of each field. This isn't a problem for abstractions that are

unlikely to change, such as dates. But tuples are less appropriate if the structure isn't final and is likely to change in the future. For such cases, you're better off using maps.

Using a map to represent a structure with named fields is simple. The following snippet demonstrates how you can create and use an entry instance:

```
iex(1)> entry = %{date: {2013, 12, 19}, title: "Dentist"}

iex(2)> entry.date
{2013, 12, 19}

iex(3)> entry.title
"Dentist"
```

You can immediately adapt your code to represent entries with maps. As it turns out, this change is extremely simple. All you need to do is change the code of the `TodoList .add_entry` function to accept two arguments: a to-do list instance, and a map that describes an entry. The new version is presented in the following listing.

**Listing 4.6. Representing entries with maps (todo_entry_map.ex)**

```
defmodule TodoList do
  ...

  def add_entry(todo_list, entry) do
    MultiDict.add(todo_list, entry.date, entry)
  end

  ...
end
```

That was simple! You assume an entry is a map and add it to `MultiDict` using its date field as a key.

Let's see this in action. To add a new entry, clients now must provide a map:

```
iex(1)> todo_list = TodoList.new |>
          TodoList.add_entry(%{date: {2013, 12, 19}, title:
"Dentist"})
```

The client code is obviously more verbose, because it must provide field names. But because entries are now structured in a map, data

retrieval is improved. The `TodoList .entries/2` function now returns complete entries, not just their titles:

```
iex(2)> TodoList.entries(todo_list, {2013, 12, 19})
[%{date: {2013, 12, 19}, title: "Dentist"}]
```

The current implementation of `TodoList` relies on `HashDict`. This means that at runtime, it's impossible to make a distinction between a `HashDict` and a `TodoList` instance. In some situations, you may want to define and enforce a more precise structure definition. For such cases, Elixir provides a construct called *structs*.

### 4.1.4. Abstracting with structs

Let's say you need to perform various arithmetic operations on fractions (a/b). This could be done efficiently if you had a corresponding abstraction. The following snippet demonstrates how could such an abstraction be used:

```
$ iex fraction.ex

iex(1)> Fraction.add(Fraction.new(1, 2), Fraction.new(1, 4)) |>
          Fraction.value
0.75
```

Here you sum one half (1/2) with one quarter (1/4) and return the numerical value of the resulting fraction. A fraction is created using `Fraction.new/2` and then passed to various other functions that know how to work with it.

Notice that there's no notion of how the fraction is represented. A client instantiates the abstraction and passes it on to another function from the corresponding module.

How can you implement this? There are many approaches, such as relying on plain tuples or using maps. In addition, Elixir provides a facility called *structs* that allows you to specify the abstraction structure up front and bind it to a module. Each module may define only one struct, which can then be used to create new instances and pattern-match on them.

In this case, a fraction has a well-defined structure, so you can use `struct` to specify and enforce data correctness. Let's see this in action. To define a struct, you use the `defstruct` macro, as shown next.

**Listing 4.7. Defining a structure (fraction.ex)**

```
defmodule Fraction do
  defstruct a: nil, b: nil

  ...
end
```

A keyword list provided to `defstruct` defines the struct's fields together with their initial values. You can now instantiate a struct using this special syntax:

```
iex(1)> one_half = %Fraction{a: 1, b: 2}
%Fraction{a: 1, b: 2}
```

Notice how a struct bears the name of the module it's defined in. There is a tight relation between structs and modules. A struct may exist only in a module, and a single module can define only one struct.

Underneath, a struct instance is a special kind of a map. Therefore, individual fields are accessed just like maps:

```
iex(2)> one_half.a
1

iex(3)> one_half.b
2
```

The nice thing about structs is that you can pattern-match on them:

```
iex(4)> %Fraction{a: a, b: b} = one_half
%Fraction{a: 1, b: 2}

iex(5)> a
1

iex(6)> b
2
```

This makes it possible to assert that some variable is really a struct:

```
iex(6)> %Fraction{} = one_half
%Fraction{a: 1, b: 2}
```
Successful match

```
iex(7)> %Fraction{} = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}
```
Map doesn't match a struct

Here you use a `%Fraction{}` pattern that matches any `Fraction` struct, regardless of its contents. Pattern matching with structs works similarly to that of maps. This means in a pattern match, you need to specify only the fields you're interested in, ignoring all other fields.

Updating a struct works similarly to the way it works with maps:

```
iex(8)> one_quarter = %Fraction{one_half | b: 4}
%Fraction{a: 1, b: 4}
```

This code creates a new struct instance based on the original one (`one_half`), changing the value of the field `b` to 4.

Armed with this knowledge, let's add some functionality to the `Fraction` abstraction. First you need to provide the instantiation function.

**Listing 4.8. Instantiating a fraction (fraction.ex)**

```
defmodule Fraction do
  ...

  def new(a, b) do
    %Fraction{a: a, b: b}
  end

  ...
end
```

This is a simple wrapper around the `%Fraction{}` syntax. It makes the client code clearer and less coupled with the fact that structs are used.

Next, let's implement a `Fraction.value/1` function that returns a decimal representation of the fraction.

**Listing 4.9. Calculating the fraction value (fraction.ex)**

```
defmodule Fraction do
  ...

  def value(%Fraction{a: a, b: b}) do    <-- Matches a fraction
    a / b
  end

  ...
end
```

`value/1` matches a fraction, taking its fields into individual variables and using them to compute the final result. The benefit of pattern matching is that input type is enforced. If you pass anything that isn't a fraction instance, you'll get a match error.

Instead of decomposing fields into variables, you could also use dot notation:

```
def value(fraction) do
  fraction.a / fraction.b
end
```

The code is arguably clearer, but it will run slightly more slowly than the previous case where you read all fields in a match. This performance penalty shouldn't make much of a difference in most situations, so you're advised to choose the approach you find more readable.

One final thing left to do is to implement the `add` function.

**Listing 4.10. Adding two fractions (fraction.ex)**

```
defmodule Fraction
  ....

  def add(
    %Fraction{a: a1, b: b1},
    %Fraction{a: a2, b: b2}
  ) do
    new(
      a1 * b2 + a2 * b1,
      b2 * b1
    )
  end

  ....
end
```

You can now test your fraction:

```
iex(1)> Fraction.add(Fraction.new(1, 2), Fraction.new(1, 4)) |>
          Fraction.value
0.75
```

The code works as expected. By representing fractions with a struct, you can provide the definition of your type, listing all fields and their default values. Furthermore, it's possible to distinguish struct instances from any other data type. This allows you to place `%Fraction{}` matches in function arguments, thus asserting that you only accept fraction instances.

**Structs vs. maps**

You should always be aware that structs are in reality just maps, and therefore they have the same characteristics with respect to performance and memory usage. But a struct instance receives special treatment. Many things that can be done with maps don't work with structs. For example, you can't call the `Enum` function on a struct:

```
iex(1)> one_half = Fraction.new(1, 2)

iex(2)> Enum.to_list(one_half)
** (Protocol.UndefinedError) protocol Enumerable not implemented for
  %Fraction{a: 1, b: 2}
```

Remember, a struct is a functional abstraction and should therefore behave according to the implementation of the module where it's defined. In the case of the `Fraction` abstraction, you must define whether `Fraction` is enumerable and, if so, in what way. If this isn't done, `Fraction` isn't an enumerable, so you can't call `Enum` functions on it.

In contrast, a plain map is an enumerable, so you can convert it to a list:

```
iex(3)> Enum.to_list(%{a: 1, b: 2})
[a: 1, b: 2]
```

On the other hand, because structs are maps, directly calling `Map` functions works:

```
iex(4)> Map.to_list(one_half)
```

```
[__struct__: Fraction, a: 1, b: 2]
```

Notice the `__struct__: Fraction` bit. This key-value pair is automatically included in each struct. It helps Elixir distinguish structs from plain maps and perform proper runtime dispatches from within polymorphic generic code. You'll learn more about this later when we describe protocols.

The `struct` field has an important consequence on pattern matching. A struct pattern can't match a plain map:

```
iex(5)> %Fraction{} = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}
```

But a plain map pattern can match a struct:

```
iex(5)> %{a: a, b: b} = %Fraction{a: 1, b: 2}
%Fraction{a: 1, b: 2}

iex(6)> a
1

iex(7)> b
2
```

This is due to the way pattern matching works with maps. Remember, all fields from the pattern must exist in the matched term. When matching a map to a struct pattern, this isn't the case, because `%Fraction{}` contains the field `__struct__` that isn't present in the map being matched.

The opposite works, because you match a struct to the `%{a: a, b: b}` pattern. Because all these fields exist in the `Fraction` struct, the match is successful.

**Records**

In addition to maps and structs, there is another way to structure data: *records*. This is a facility that let's you use tuples and still be able to access individual elements by name. Records can be defined using the `defrecord` and `defrecordp` macros from the `Record` module (http://elixir-lang.org/docs/stable/elixir/Record.html).

Given that they're essentially tuples, records should be faster than maps (although the difference usually doesn't make a significant difference in the grand scheme of things). On the flip side, usage is more verbose, and it's not possible to access fields by name dynamically.

Records are present mostly for historical reasons. Before maps appeared, records were one of the main tools for structuring data. In fact, many libraries from the Erlang ecosystem use records as their interface. One such example is `xmerl`—a library for XML manipulation.

If you need to interface an Erlang library using a record defined in that library, you must import that record into Elixir and define it as a record. This can be done with the `Record.extract/2` function in conjunction with the `defrecord` macro. This idiom isn't required often, so records aren't demonstrated here. Still, it may be useful to keep this information in the back of your head and research it if the need arises.

### 4.1.5. Data transparency

The modules you've devised so far are abstractions, because clients aren't aware of their implementation details. For example, as a client, you call `Fraction.new/2` to create an instance of the abstraction and then send that instance back to some other function from that same module.

But the entire data structure is always visible. As a client, you can obtain individual fraction values, even if this was not intended by the library developer.

It's important to be aware that data in Elixir is always transparent. Clients can read individual informations from your structs (and any other data type), and there is no easy way of preventing that. In that sense, encapsulation works differently than in typical OO languages. In Elixir, modules are in charge of abstracting the data and providing operations to manipulate and query that data. But the data is never

hidden.

Let's verify this in a shell session:

```
$ iex todo_entry_map.ex

iex(1)> todo_list = TodoList.new |>
          TodoList.add_entry(%{date: {2013, 12, 19}, title: "Dentist"})

#HashDict<[
  {{2013, 12, 19}, [%{date: {2013, 12, 19}, title: "Dentist"}]}
]>
```

To-do list with a single element

Looking at the return value, you can see the entire structure of the to-do list. From the output, you can immediately tell that the to-do list is implemented using the `HashDict` structure, and you can also find out details about how individual entries are kept.

Notice how `HashDict` is printed in a special way, using `#HashDict<...>` output. This is due to the inspection mechanism in Elixir: whenever a result is printed in the shell, the function `Kernel.inspect/1` is called to transform the structure into an inspected string. For each abstraction you build, you can override the default behavior and provide your own inspected format. This is exactly what `HashDict` does, and you'll learn how to do this for your type later in this chapter when we discuss protocols.

Occasionally you may want to see the pure data structure, without this decorated output. This can be useful when you're debugging, analyzing, or reverse-engineering code. To do so, you can provide a special option to the `inspect` function:

```
iex(2)> IO.puts(inspect(todo_list, structs: false))

%{__struct__: HashDict, root: {[], [], [],
  [{2013, 12, 19}, %{date: {2013, 12, 19}, title: "Dentist"}],
  [], [], [], []}, size: 1}
```

The output now reveals the complete structure of the to-do list, and you can "see through" the `HashDict` abstraction. This demonstrates that data privacy can't be enforced in functional abstractions; you can see the naked structure of the data. Remember from chapter 2 that the only complex types are tuples, lists, and maps. Any other abstraction, such as `HashDict` or your own `TodoList`, will ultimately be built

on top of these types.

The benefit of data transparency is that the data can be easily inspected, which can be useful for debugging purposes. But as a client of a data abstraction, you shouldn't rely on its internal representation, even though it's visible to you. You shouldn't pattern-match on the internal structure or try to extract or modify individual parts of it. The reason is that a proper abstraction, such as `HashDict`, doesn't guarantee what the data will look like. The only guarantee is that the module's functions will work if you send them a properly structured instance that you already received from that same module.

One final thing you should know about data inspection is the `IO.inspect/1` function. This function prints the inspected representation of a structure to the screen and returns the structure itself. This is particularly useful when debugging a piece of code. Look at the following example:

```
iex(1)> Fraction.new(1, 4) |>
          Fraction.add(Fraction.new(1, 4)) |>
          Fraction.add(Fraction.new(1, 2)) |>
          Fraction.value
1.0
```

This code relies on the pipeline operator to perform a series of fractions operations. Let's say you want to inspect the entire structure after each step. You can easily insert the call to `IO.inspect` after every line:

```
iex(2)> Fraction.new(1, 4) |>
          IO.inspect |>
          Fraction.add(Fraction.new(1, 4)) |>
          IO.inspect |>
          Fraction.add(Fraction.new(1, 2)) |>
          IO.inspect |>
          Fraction.value

%Fraction{a: 1, b: 4}
%Fraction{a: 8, b: 16}
%Fraction{a: 32, b: 32}
```

**Output of each IO.inspect call**

This works because `IO.inspect/1` prints the data structure and then returns that same data structure unchanged.

We are now done with the basic theory behind functional abstractions, but you'll practice some more by extending the to-do list.

## 4.2. Working with hierarchical data

In this section, you'll extend the `TodoList` abstraction to provide basic CRUD support. You already have the C and R parts resolved with the `add_entry/3` and `entries/1` functions, respectively. Now you need to add support for updating and deleting entries. To do this, you must be able to uniquely identify each entry in the to-do list. So, you'll begin by adding unique ID values to each entry.

### 4.2.1. Generating IDs

When adding a new entry to the list, you'll autogenerate its ID value, using incremental integers for IDs. To implement this, you have to do a couple of things:

- You need to transform the to-do list into a struct. This is because the to-do list now has to keep two pieces of information: the entries collection and the ID value for the next entry.
- So far, when storing entries in a collection, you used the entry's date as the key. You'll change this and use the entry's ID instead. This will make it possible to quickly insert, update, and delete individual entries. Because you'll now have exactly one value per each key, you won't need the `MultiDict` abstraction anymore.

Let's start implementing this. The code in the following listing contains the module and struct definitions.

**Listing 4.11. TodoList struct (todo_crud.ex)**

```
defmodule TodoList do
  defstruct auto_id: 1, entries: HashDict.new

  def new, do: %TodoList{}

  ...
end
```

Struct that describes the to-do list

Creates a new instance

The to-do list will now be represented as the struct with two fields. The field `auto_id` contains the ID value that will be assigned to the new entry while it's being added to the structure. The field `entries` is the collection of entries. As has been mentioned, you're now using `HashDict`, and the keys are entry ID values.

During the struct definition, the default values for the `auto_id` and `entries` fields are immediately specified. Therefore, you don't have to provide these when creating a new instance. The instantiation function `new/0` creates and returns an instance of the struct.

Next, it's time to reimplement the `add_entry/2` function. It has to do more work:

- Set the ID for the entry being added.
- Add the new entry to the collection.
- Increment the `auto_id` field.

Here's the code.

**Listing 4.12. Autogenerating ID values for new entries (todo_crud.ex)**

```
defmodule TodoList do
  ...

  def add_entry(
    %TodoList{entries: entries, auto_id: auto_id} = todo_list,
    entry
  ) do
    entry = Map.put(entry, :id, auto_id)
    new_entries = HashDict.put(entries, auto_id, entry)

    %TodoList{todo_list |
      entries: new_entries,
      auto_id: auto_id + 1
    }
  end

  ...
end
```

Sets the new entry's ID

Adds the new entry the entries list

Updates the struct

A lot of things happen here, so let's take them one at a time. In the arguments specification, you state that you expect a `TodoList` instance. This instance is destructured so you have individual fields in appropriate variables. Furthermore, you keep the entire instance in a `todo_list` variable. This is an example of a compound match, explained in section 3.1.8.

In the function body, you first update the entry's `id` value with the value stored in the `auto_id` field. Notice how you use `Map.put/3` to update the entry map. This is because the input map may not contain the `id` field. Therefore, you can't use the standard `%{entry | id: auto_id}` technique, because this works only if the `id` field is already present in the map. In any case, after the entry is updated, you add it to the entries collection, keeping the result in the `new_entries` variable.

Finally, you must update the `TodoList` struct instance, setting its `entries` field to the `new_entries` collection and incrementing the `auto_id` field. Essentially, you made a complex change in the struct, modifying multiple fields as well as the input entry (because you set its `id` field).

To the external caller, the entire operation will be atomic. Either everything will happen or, in case of an error, nothing at all. This is the consequence of immutability. The effect of adding an entry is visible to others only when the `add_entry/2` function finishes and its result is taken into a variable. If something goes wrong, and you raise an error, the effect of any transformations won't be visible.

It's also worth repeating, as mentioned in chapter 2, that the new to-do list (the one returned by the `add_entry/2` function) will share as much memory as possible with the input to-do list.

With the `add_entry/2` function finished, you need to adjust the `entries/2` function. This will be more complicated because you changed the internal structure. Earlier, you kept a date-to-entries mapping. Now, entries are stored using `id` as the key, so you have to iterate through all the entries and return the ones that fall on a given date. The code is given next.

**Listing 4.13. Filtering entries for a given date (todo_crud.ex)**

```
defmodule TodoList do
  ...

  def entries(%TodoList{entries: entries}, date) do
    entries
    |> Stream.filter(fn({_, entry}) ->
         entry.date == date
       end)

    |> Enum.map(fn({_, entry}) ->
         entry
       end)
  end

  ...
end
```

Filters entries for a given date

Takes only values

This function takes advantage of the fact that `HashDict` is an enumerable. When you use a `HashDict` instance with functions from `Enum` or `Stream`, then each `HashDict` element is treated in the form of `{key, value}`.

Two transformations are performed here. First, you take only those entries that fall on a given date. After that, you have a collection of `{id, entry}` tuples (due to `HashDict` enumerable properties just mentioned). Therefore, you have to do an additional transformation and extract only the second element of each tuple.

Notice that the first transformation is done with the `Stream` module, whereas the second one uses `Enum`. As explained in chapter 3, this allows both transformations to happen in a single pass through the input collection.

Finally, you can check if your new version of the to-do list works:

```
$ iex todo_crud.ex

iex(1)> todo_list = TodoList.new |>
          TodoList.add_entry(
              %{date: {2013, 12, 19}, title: "Dentist"}
          ) |>

          TodoList.add_entry(
            %{date: {2013, 12, 20}, title: "Shopping"}
          ) |>
```

```
          TodoList.add_entry(
            %{date: {2013, 12, 19}, title: "Movies"}
          )

iex(2)> TodoList.entries(todo_list, {2013, 12, 19})

[%{date: {2013, 12, 19}, id: 3, title: "Movies"},
 %{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

This works as expected, and you can even see the ID value for each entry. Also note that the interface of the `TodoList` module is the same as the previous version. You've made a number of internal modifications, changed the data representation, and practically rewritten the entire module. And yet the module's clients don't need to be altered, because you kept the same interface for your functions.

Of course, this is nothing revolutionary—it's a classical benefit of abstracting the data. But it demonstrates how to construct and reason about higher-level types when working with stateless modules and immutable data.

As an added bonus of this refactoring, you've added runtime type checks to the functions `add_entry/2` and `entries/2`. Both functions now explicitly check, via pattern matching, that the first argument is an instance of the `TodoList` struct.

### 4.2.2. Updating entries

Now that your entries have ID values, you can add additional modifier operations. Let's implement the `update_entry` operation, which can be used to modify a single entry in the to-do list.

The first question is, how will the `update_entry` function be used? There are two possible options:

1. The function will accept an ID value of the entry and an updater lambda. This will work similarly to `HashDict.update`. The lambda will receive the original entry and return its modified version.

2. The function will accept an entry map. If an entry with the same ID exists in the `entries` collection, it will be replaced.

Another issue is what to do if an entry with a given ID doesn't exist. You can either do nothing or raise an error. In this example, you'll use option 1, the updater lambda approach, and you won't raise an error if the entry with a given ID doesn't exist.

The following snippet illustrates the usage. Here, you modify the date of an entry that has an ID value of 1:

```
iex(1)> TodoList.update_entry(
          todo_list,
          1,
          &Map.put(&1, :date, {2013, 12, 20})
        )
```

ID of the entry to be modified

Modifies an entry date

The implementation is presented in the following listing.

**Listing 4.14. Updating an entry (todo_crud.ex)**

```
defmodule TodoList do
  ...
  def update_entry(
    %TodoList{entries: entries} = todo_list,
    entry_id,
    updater_fun
  ) do
    case entries[entry_id] do
      nil -> todo_list

      old_entry ->
        new_entry = updater_fun.(old_entry)
        new_entries = HashDict.put(entries, new_entry.id, new_entry)
        %TodoList{todo_list | entries: new_entries}
    end
  end

  ...
end
```

No entry—returns the unchanged list

Entry exists—performs the update and returns the modified list

Let's break down what happens here. In the argument specifiers for `update_entry/3`, you pattern-match a `TodoList` struct and store its `entries` field in a separate value. In addition, the entire instance is taken into a `todo_list` variable.

Then you look up the entry with the given ID and branch on the result. If the entry doesn't exist, you return the original version of the

list.

Otherwise, you have to call the updater lambda to get the modified entry. Then you store the modified entry into the entries collection. Finally, you store the modified entries collection in the `TodoList` instance and return that instance.

Notice how the `case` construct is used for conditional logic. You may wonder why this approach is chosen over multiclause functions, described in chapter 3. This is a matter of personal judgment in a specific situation. In this case, you have a fairly simple conditional, so it seems more compact and understandable to write it all at once without breaking the code into multiple functions.

#### Fun with Pattern Matching

`update_entry/3` works fine, but it's not quite bulletproof. The updater lambda can return any data type, possibly corrupting the entire structure. You can make some assertions about this with a little help from pattern matching, as shown in the following snippet:

```
new_entry = %{} = updater_fun.(old_entry)
```

Here you use a nested match, as explained in chapter 3: you require that the result of the updater lambda be a map. If that fails, an error will be raised. Otherwise, you'll take the entire result into the `new_entry` variable.

You can go a step further and assert that the ID value of the entry has not been changed in the lambda:

```
old_entry_id = old_entry.id
new_entry = %{id: ^old_entry_id} = updater_fun.(old_entry)
```

Here you store the ID of the old entry in the separate variable. Then you check that the result of the updater lambda has that same ID value. Recall from chapter 3 that `^var` in a pattern match means you're matching on the value of the variable. In this case, you match on the value stored in the `old_entry_id` variable. If the updater lambda returns an entry with a different ID, the match will fail, and

an error will be raised.

You can also use pattern matching to provide an alternative update interface. The following snippet depicts the idea:

```
def update_entry(todo_list, %{} = new_entry) do
  update_entry(todo_list, new_entry.id, fn(_) -> new_entry end)
end
```

Here you define an `update_entry/2` function that expects a to-do list, and an entry that is a map. It delegates to `update_entry/3`, which you just implemented. The updater lambda ignores the old value and returns the new entry. Recall from chapter 3 that functions with the same name but different arities are different functions. This means you now have two distinct `update_entry` functions. Which one will be called depends on the number of arguments passed.

Finally, note that this abstraction is still overly vague. When clients provide entries via `TodoList.add_entry/2` and `TodoList.update_entry/2`, there are no restrictions on what the entries should contain. These functions accept any kind of map. To make this abstraction more restrictive, you could introduce a dedicated struct for the entry (for example, a `TodoEntry`) and then use pattern matching to enforce that each entry is in fact an instance of that struct.

### 4.2.3. Immutable hierarchical updates

You may not have noticed, but in the previous example, you performed a deep update of an immutable hierarchy. Let's break down what happens when you call `TodoList.update_entry(todo_list, id, updater_lambda)`:

1. You take the entries collection into a separate variable.
2. You take the target entry from that collection into a separate variable.
3. You call the updater that returns the modified version of the

entry to you.

4. You call `HashDict.put` to put the modified entry into the entries collection.

5. You return the new version of the to-do list, which contains the new entries collection.

Notice that steps 3, 4, and 5 are the ones where you transform data. Each of these steps creates a new variable that contains the transformed data. In each subsequent step, you take this data and update its container, again by creating a transformed version of it.

This is how you work with immutable data structures. If you have hierarchical data, you can't directly modify part of it that resides deep in its tree. Instead, you have to walk down the tree to the particular part that needs to be modified and then transform it and all of its ancestors. The result is a copy of the entire model (in this case, the to-do list). As mentioned, the two versions—new and previous—will share as much memory as possible.

If you divide these operations into many smaller functions, the implementation is fairly simple and straightforward. Each function is responsible for modifying part of the hierarchy. If it needs to change something that is deeper in the tree, it finds the topmost ancestor and then delegates to another function to perform the rest of the work. In this way, you're breaking down the walk through the tree into many smaller functions, each of which is in charge of a particular level of abstraction.

#### Provided helpers

Although the technique presented works, it may become cumbersome for deeper hierarchies. Remember, to update an element deep in the hierarchy, you have to walk to that element and then update all of its parents. To simplify this, Elixir offers support for more elegant deep hierarchical updates.

Let's look at a basic example. Internally, the to-do structure is a simple `HashDict`, where keys are IDs and values are plain maps consisting of

fields. Let's create one such naked structure:

```
iex(1)> todo_list = [
          {1, %{date: {2013, 12, 19}, title: "Dentist"}},
          {2, %{date: {2013, 12, 20}, title: "Shopping"}},
          {3, %{date: {2013, 12, 19}, title: "Movies"}}
        ] |> Enum.into(HashDict.new)

#HashDict<[
  {2, %{date: {2013, 12, 20}, title: "Shopping"}},
  {3, %{date: {2013, 12, 19}, title: "Movies"}},
  {1, %{date: {2013, 12, 19}, title: "Dentist"}}
]>
```

Here you create a list of two-element key-value tuples and then use `Enum.into/1` to transform this into a `HashDict`.

Now, let's say you change your mind and want to go to the theater instead of a movie. The original structure can be modified elegantly using the `Kernel.put_in/2` macro:

```
iex(2)> put_in(todo_list[3][:title], "Theater")      <-- Hierarchical update

#HashDict<[
  {2, %{date: {2013, 12, 20}, title: "Shopping"}},
  {3, %{date: {2013, 12, 19}, title: "Theater"}},    <-- Entry title is updated
  {1, %{date: {2013, 12, 19}, title: "Dentist"}}
]>
```

What happened here? Internally, `put_in/2` does something similar to what you did. It walks recursively to the desired element, transforms it, and then updates all the parents. Notice that this is still an immutable operation, meaning the original structure is left intact, and you have to take the result to a variable.

To be able to do a recursive walk, `put_in/2` needs to receive source data and a path to the target element. In the example, the source is provided as `todo_list` and the path is specified as `[3][:title]`. The macro `put_in/2` then walks down that path, rebuilding the new hierarchy on the way up.

For this to work, there must be a generic way of accessing a field of each element. This is provided via a mechanism called *protocols*, which we'll explain in section 4.3. Long story short, protocols allow you to invoke polymorphic actions. The macro `put_in/2` relies on

the `Access` protocol, which allows you to work with key-value structures such as `HashDict` and maps.

It's also worth noting that Elixir provides similar alternatives for data retrieval and updates in the form of the `get_in/2`, `update_in/2`, and `get_and_update_in/2` macros. The fact that these are macros means the path you provide is evaluated at compile-time and can't be built dynamically.

If you need to construct paths at runtime, there are equivalent functions that accept data and path as separate arguments. For example, `put_in` can also be used as follows:

```
iex(3)> path = [3, :title]
iex(4)> put_in(todo_list, path, "Theater")
```

Using a path constructed at runtime

**Exercise: deleting an entry**

Your `TodoList` module is almost complete. You've already implemented create (`add_entry/2`), retrieve (`entries/2`), and update (`update_entry/3`) operations. The last thing remaining is the `delete_entry/2` operation. This is straightforward, and it's left for you to do as an exercise. If you get stuck, the solution is provided in the source file todo_crud.ex.

### 4.2.4. Iterative updates

So far, you've been doing updates manually, one at the time. Now it's time to see how to implement iterative updates. Imagine that you have a raw list describing the entries:

```
$ iex todo_builder.ex

iex(1)> entries = [
          %{date: {2013, 12, 19}, title: "Dentist"},
          %{date: {2013, 12, 20}, title: "Shopping"},
          %{date: {2013, 12, 19}, title: "Movies"}
        ]
```

Now you want to create an instance of the to-do list that contains all of these entries:

```
iex(2)> todo_list = TodoList.new(entries)

iex(3)> TodoList.entries(todo_list, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 3, title: "Movies"},
 %{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

It's obvious that the function `new/1` performs an iterative build of the to-do list. How can you implement such a function? As it turns out, this is simple, as you can see in the following listing.

**Listing 4.15. Iteratively building the to-do list (todo_builder.ex)**

```
defmodule TodoList do
  ...

  def new(entries \\ []) do
    Enum.reduce(
      entries,
      %TodoList{},
      fn(entry, todo_list_acc) ->
        add_entry(todo_list_acc, entry)
      end
    )
  end

  ...
end
```

Initial accumulator value

Iteratively updates the accumulator

To build the to-do list iteratively, you're relying on `Enum.reduce/3`. Recall from chapter 3 that `reduce` is used to transform something enumerable to anything else. In this case, you're transforming a raw list of `Entry` instances into an instance of the `TodoList` struct. Therefore, you call `Enum.reduce/3`, passing the input list as the first argument, the new structure instance as the second argument (the initial accumulator value), and the lambda that is called in each step.

The lambda is called for each entry in the input list. Its task is to add the entry to the current accumulator (`TodoList` struct) and return the new accumulator value. To do this, the lambda delegates to the already-present `add_entry/2` function, reversing the argument order. The arguments need to be reversed because `Enum.reduce/3` calls the lambda, passing the iterated element (entry) and accumulator (`TodoList` struct). In contrast, `add_entry` accepts a struct and an entry.

Notice that you can make the lambda definition more compact with

the help of the capture operator:

```
def new(entries \\ []) do
  Enum.reduce(
    entries,
    %TodoList{},
    &add_entry(&2, &1)
  )
end
```

Reverses the order of arguments and delegates to add_entry/2

Whether you prefer this version or the previous one is entirely up to your personal taste. The capture version is definitely shorter, but it's arguably more cryptic.

### 4.2.5. Exercise: importing from a file

Now it's time for you to practice a bit. In this exercise, you'll create a `TodoList` instance from the comma-separated file.

Assume that you have a file todos.csv in the current folder. Each line in the file describes a single to-do entry:

```
2013/12/19,Dentist
2013/12/20,Shopping
2013/12/19,Movies
```

Your task is to create the additional module `TodoList.CsvImporter` that can be used to create a `TodoList` instance from the file contents:

```
iex(1)> todo_list = TodoList.CsvImporter.import("todos.csv")
```

To simplify the task, assume that the file is always available and in the correct format. Also assume that the comma character doesn't appear in the entry title.

This is generally not hard to do, but it might require some cracking and experimenting. Here are a couple of hints that will lead you in the right direction.

First, create a single file with the following layout:

```
defmodule TodoList do
  ...
end

defmodule TodoList.CsvImporter do
  ...
end
```

Always work in small steps. Implement part of the calculation, and then print the result to the screen using `IO.inspect/1`. I can't stress enough how important this is. This task requires some data pipelining. Working in small steps will allow you to move gradually and verify that you're on the right track.

The general steps you should undertake are as follows:

**1**. Open a file and go through it, removing `\n` from each line. Hint: use `File.stream!/1`, `Stream.map/2`, and `String.replace/2`. You did this in chapter 3, when we talked about streams, in the example where you filtered lines longer than 80 characters.

**2**. Parse each line obtained from the previous step into a raw tuple in the form `{{year, month, date}, title}`. Hint: you have to split each line using `String.split/2`. Then further split the first element (date), and extract the date parts. `String.split/2` returns a list of strings separated by the given token. When you split the date field, you'll have to additionally convert each date part into a number. Use `String.to_integer/1` for this.

**3**. Once you have the raw tuple, create a map that represents the entry.

**4**. The output of step 3 should be an enumerable that consists of maps. Pass that enumerable to the `TodoList.new/1` function that you recently implemented.

In each of these steps, you receive an enumerable as an input, transform each element, and pass the resulting enumerable forward to

the next step. In the final step, the resulting enumerable is passed to the already-implemented `TodoList.new/1`, and the to-do list is created.

If you work in small steps, it's harder to get lost. For example, you can start by opening a file and printing each line to the screen. Then try to remove the trailing newline from each line and print them to the screen, and so on.

While transforming the data in each step, you can work with `Enum` functions or functions from the `Stream` module. It will probably be simpler to start with eager functions from the `Enum` module and get the entire thing to work. Then you should try to replace as many of the `Enum` functions as possible with their `Stream` counterparts. Recall from chapter 3 that the `Stream` functions are lazy and composable, which results in a single iteration pass. If you get lost, the solution is provided in the file todo_import.ex.

In the meantime, we're almost done with the exploration of higher-level data abstractions. The final topic we'll briefly discuss is the Elixir way of doing polymorphism.

## 4.3. Polymorphism with protocols

*Polymorphism* is a runtime decision about which code to execute, based on the nature of the input data. In Elixir, the basic (but not the only) way of doing this is by using the language feature called *protocols*.

Before looking at protocols, let's see them in action. You've already seen polymorphic code. For example, the entire `Enum` module is generic code that works on anything *enumerable*, as the following snippet illustrates:

```
Enum.each([1, 2, 3], &IO.puts/1)
Enum.each(1..3, &IO.puts/1)
Enum.each(hash_dict, &IO.puts/1)
```

Notice how you use the same `Enum.each/2` function, sending it different data structures: a list, a range, and a `HashDict` instance. How does `Enum.each/2` know how to walk each structure? It doesn't. The code in `Enum.each/2` is generic and relies on a contract. This contract, called a *protocol*, must be implemented for each data type you wish to use with `Enum` functions. This is roughly similar to abstract interfaces from OO, with a slight twist. Let's see how to define and use protocols.

### 4.3.1. Protocol basics

A *protocol* is a module in which you declare functions without implementing them. Consider it a rough equivalent of an OO interface. The generic logic relies on the protocol and calls its functions. Then you can provide a concrete implementation of the protocol for different data types.

Let's look at an example. The protocol `String.Chars` is provided by the Elixir standard libraries and is used to convert data to a binary string. This is how the protocol is defined in the Elixir source:

```
defprotocol String.Chars do     <-- Definition of the protocol
  def to_string(thing)          <-- Declaration of protocol functions
end
```

This resembles the module definition, with the notable difference that functions are declared but not implemented.

Notice the first argument of the function (the `thing`). At runtime, the type of this argument determines the implementation that's called. Let's see this in action. Elixir already implements the protocol for atoms, numbers, and some other data types. Therefore, you can issue following calls:

```
iex(1)> String.Chars.to_string(1)
"1"
iex(2)> String.Chars.to_string(:an_atom)
"an_atom"
```

If the protocol isn't implemented for the given data type, an error is

raised:

```
iex(3)> String.Chars.to_string(TodoList.new)
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

Usually you don't need to call the protocol function directly. More often, there is generic code that relies on the protocol. In the case of `String.Chars`, this is the auto-imported function `Kernel.to_string/1`:

```
iex(4)> to_string(1)
"1"

iex(5)> to_string(:an_atom)
"an_atom"

iex(6)> to_string(TodoList.new)
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

As you can see, the behavior of `to_string/1` is exactly the same as that of `String.Chars.to_string/1`. This is because `Kernel.to_string/1` delegates to the `String.Chars` implementation.

In addition, you can send anything that implements `String.Chars` to `IO.puts/1`:

```
iex(7)> IO.puts(1)
1

iex(8)> IO.puts(:an_atom)
an_atom

iex(9)> IO.puts(TodoList.new)
** (Protocol.UndefinedError) protocol String.Chars not implemented
```

As you can see, an instance of the `TodoList` isn't printable because `String.Chars` isn't implemented for the corresponding type.

### 4.3.2. Implementing a protocol

How do you implement a protocol for a specific type? Let's refer to the Elixir source again. The following snippet implements

`String.Chars` for integers:

```
defimpl String.Chars, for: Integer do
  def to_string(thing) do
    Integer.to_string(thing)
  end
end
```

You start the implementation by calling the `defimpl` macro. Then you specify which protocol to implement and the corresponding data type. Finally, the `do/end` block contains the implementation of each protocol function. In the example, the implementation delegates to the existing standard library function `Integer.to_string/1`.

The `for: Type` part deserves some explanation. The type is an atom and can be any of following aliases: `Tuple`, `Atom`, `List`, `Map`, `BitString`, `Integer`, `Float`, `Function`, `PID`, `Port`, or `Reference`. These values correspond to built-in Elixir types.

In addition, the alias `Any` is allowed, which makes it possible to specify a fallback implementation. If a protocol isn't defined for a given type, an error will be raised, unless a fallback to any is specified in the protocol definition and an `Any` implementation exists. Refer to the `defprotocol` documentation (http://elixir-lang.org/docs/stable/elixir/Kernel.html#defprotocol/2) for details.

Finally, and most important, the type can be any other arbitrary alias (but not a regular, simple atom):

```
defimpl String.Chars, for: SomeAtom do
  ...
end
```

This implementation will be called if the first argument of the protocol function (the *thing*) is a struct defined in the corresponding module.

For example, you can implement `String.Chars` for `TodoList`. Let's do this:

```
iex(1)> defimpl String.Chars, for: TodoList do
          def to_string(_) do
            "#TodoList"
```

```
    end
  end
```

Now you can pass a to-do list instance to `IO.puts/1`:

```
iex(2)> IO.puts(TodoList.new)
#TodoList
```

It's important to notice that the protocol implementation doesn't need to be part of any module. This has powerful consequences: you can implement a protocol for a type even if you can't modify the type's source code. You can place the protocol implementation anywhere in your own code, and the runtime will be able to take advantage of it.

**Protocols and performance**

Due to its inner workings, a protocol dispatch may in some cases be significantly slower than a direct function call. The reason is the runtime discovery of the implementation that needs to be invoked. Elixir provides a way of speeding things up in form of a *protocol consolidation*. A consolidation essentially analyzes your project and compiles the dispatch map, thus optimizing lookup operations. We'll demonstrate this later, in chapter 13, when you build a release of a system.

### 4.3.3. Built-in protocols

Elixir comes with some predefined protocols. It's best to consult the online documentation (http://elixir-lang.org/docs/stable/elixir/protocols_list.html) for the complete reference, but let's mention some of the more important ones.

You've already seen `String.Chars`, which specifies the contract for converting data into a binary string. There is also the `List.Chars` protocol, which converts input data to a character string (a list of characters).

If you want to control how your structure is printed in the debug

output (via the `inspect` function), you can implement the `Inspect` protocol.

The `Access` protocol makes it possible to directly retrieve a value from the data structure. If you implement `Access` for your own structure, you'll be able to use the `[]` syntax. The `[]` operator is specifically supported by the compiler, which turns the construct `container[key]` into `Access.get(container, key)`. This is why you can do the following:

```
dict = HashDict.new(...)
value = dict[key]
```

Possible because HashDict implements the Access protocol

Moreover, implementing `Access` allows your structure to work with `put_in`, `get_in`, and friends, mentioned in section 4.2.3.

Arguably the most important protocol is `Enumerable`. By implementing it, you can make your data structure *enumerable*. This means you can use all the functions from the `Enum` and `Stream` modules for free! This is probably the best demonstration of protocol usefulness. Both `Enum` and `Stream` are generic modules that offer many useful functions, which can work on your custom data structures as soon as you implement the `Enumerable` protocol.

Closely related to enumeration is the `Collectable` protocol. Recall from chapter 3 that a collectable structure is one that you can repeatedly add elements to. A collectable can be used with comprehensions to collect results or with `Enum.into/2` to transfer elements of one structure (enumerable) to another (collectable).

And of course, you can define your own protocols and implement them for any available data structure (your own or someone else's). See the `Kernel.defprotocol/2` documentation for more information.

#### Collectable to-do list

Let's look at a more involved example. You'll make your to-do list collectable so that you can use it as a comprehension target. This is a slightly more advanced example, so don't worry if you don't get every

detail in the first go.

To make the abstraction collectable, you have to implement the corresponding protocol:

```
defimpl Collectable, for: TodoList do
  def into(original) do
    {original, &into_callback/2}
  end

  defp into_callback(todo_list, {:cont, entry}) do
    TodoList.add_entry(todo_list, entry)
  end

  defp into_callback(todo_list, :done), do: todo_list
  defp into_callback(todo_list, :halt), do: :ok
end
```

Returns the appender lambda

Appender implementation

The exported function `into/1` is called by the generic code (comprehensions, for example). Here you provide the implementation that returns the appender lambda. This appender lambda is then repeatedly invoked by the generic code to append each element to your data structure.

The appender function receives a to-do list and an instruction hint. If you receive `{:cont, entry}`, then you must add a new entry. If you receive `:done`, then you return the list, which at this point contains all appended elements. Finally, `:halt` indicates that the operation has been canceled, and the return value is ignored.

Let's see this in action. Copy/paste the previous code into the shell, and then try the following:

```
iex(1)> entries = [
          %{date: {2013, 12, 19}, title: "Dentist"},
          %{date: {2013, 12, 20}, title: "Shopping"},
          %{date: {2013, 12, 19}, title: "Movies"}
        ]

iex(2)> for entry <- entries, into: TodoList.new, do: entry
%TodoList{...}
```

Collecting into a TodoList

By implementing the `Collectable` protocol, you essentially adapt the `TodoList` abstraction to any generic code that relies on the that protocol, such as comprehensions and `Enum.into/2`.

## 4.4. Summary

In this chapter, you've learned how to create your own higher-level data abstractions. The most important points to remember are the following:

- A module is used to create a data abstraction. A module's functions create, manipulate, and query data. Clients can inspect the entire structure but shouldn't rely on it.
- Maps can be used to group different fields together in a single structure.
- Structs are special kind of maps that allow you to define data abstractions related to a module.
- Polymorphism can be implemented with protocols. A protocol defines an interface that is used by the generic logic. You can then provide specific protocol implementations for a data type.

This chapter concludes our basic tour of the language. There is much more to Elixir than we've covered so far, but the rest of the book focuses more on BEAM. You'll learn how to harness its power using Elixir as the programming interface, and we'll explain additional language features when the need arises.

# Part 2. The platform

High availability is the main focus of Elixir/Erlang systems. The main tool for achieving this property in Erlang-powered systems is concurrency. In this part of the book, you'll learn how concurrency works and how it can help you build reliable systems. We begin with an introduction to concurrency in BEAM in chapter 5. Building on this, in chapter 6 you'll learn about OTP and generic server processes, which can simplify implementation of typical concurrent idioms. In chapter 7, you'll see an example of a more involved concurrent system. Then, in chapter 8, we present the basic error-detection mechanism, with a special focus on detecting errors in concurrent systems via supervisors. Going deeper, in chapter 9 you'll learn how to build supervision trees to minimize negative effects of errors. Finally, chapter 10 discusses how to efficiently share a system-wide state via ETS tables.

# Chapter 5. Concurrency primitives

*This chapter covers*

- Understanding BEAM concurrency principles
- Working with processes
- Working with stateful server processes
- Runtime considerations

Now that you have sufficient knowledge of Elixir and functional programming idioms, it's time to turn our attention to the Erlang platform. We'll spend some time exploring BEAM concurrency, a feature that plays a central role in Elixir's and Erlang's support for scalability, fault-tolerance, and distribution. In this chapter, we'll start our tour of BEAM concurrency by looking at basic techniques and tools. Before beginning with lower-level details, let's take a look at higher-level principles.

## 5.1. Principles

Erlang is all about writing highly available systems—systems that run forever and are always able to meaningfully respond to client requests. To make your system highly available, you have to tackle following challenges:

- Minimize, isolate, and recover from the effects of runtime errors (fault tolerance).
- Handle a load increase by adding more hardware resources without changing or redeploying the code (scalability).
- Run your system on multiple machines so that others can take over if one machine crashes (distribution).

If you address these challenges, your systems can constantly provide service with minimal downtime and failures. Concurrency plays an important role in achieving high availability. In BEAM, a unit of concurrency is a *process*: a basic building block that makes it possible

to build scalable, fault-tolerant, distributed systems.

**Note**

A BEAM process shouldn't be confused with an OS process. As you're about to learn, BEAM processes are much lighter and cheaper than OS processes. Because this book deals mostly with BEAM, in the remaining text the term *process* refers to a BEAM process.

In production, a typical server system must handle many simultaneous requests from many different clients, maintain a shared state (for example, caches, user session data, and server-wide data), and run some additional background processing jobs. For the server to work normally, all of these tasks should run reasonably fast and be reliable enough.

Because many tasks are pending simultaneously, it's imperative to execute them in parallel as much as possible, thus taking advantage of all available CPU resources. For example, it's extremely bad if long processing of one request blocks all other pending requests and background jobs. Such behavior can lead to a constant increase in the request queue, and the system can become unresponsive.

Moreover, tasks should run as isolated from each other as possible. You don't want an unhandled exception in one request handler to crash another unrelated request handler, a background job, or, especially, the entire server. You also don't want a crashing task to leave behind an inconsistent memory state, which might later compromise another task.

That is exactly what the BEAM concurrency model does for us. Processes help us run things in parallel, allowing us to achieve *scalability*—the ability to address the load increase by adding more hardware power that the system automatically takes advantage of.

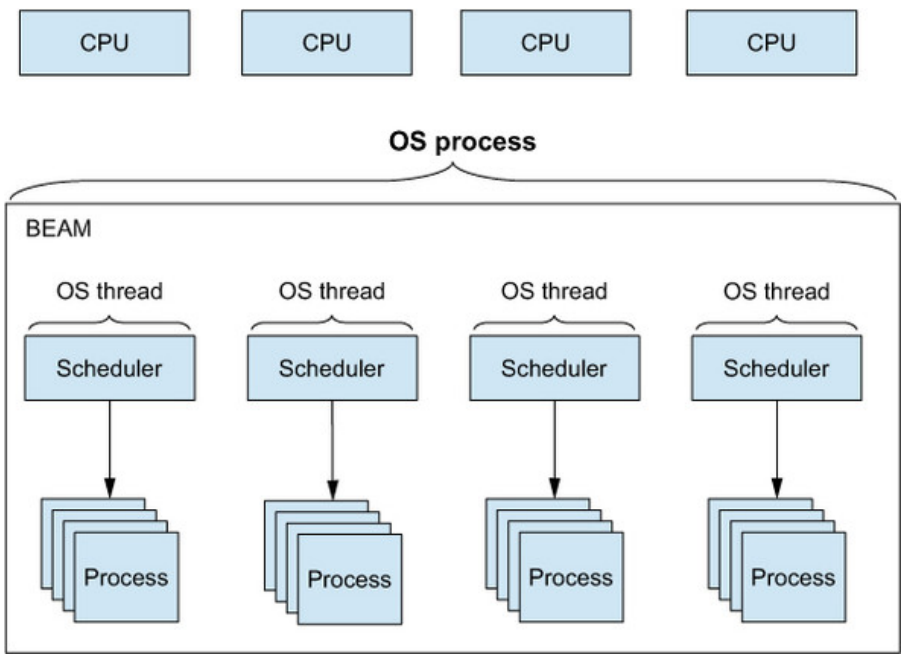
Processes also ensure isolation, which in turn gives us *fault tolerance*—the ability to localize and limit the impact of unexpected runtime errors that inevitably occur. If you can localize exceptions and recover from them, you can implement a system that truly never stops, even

when unexpected errors occur.

In BEAM, a process is a concurrent thread of execution. Two processes run concurrently and may therefore run in parallel, assuming at least two CPU cores are available. Unlike OS processes or threads, BEAM processes are lightweight concurrent entities handled by the VM, which uses its own scheduler to manage their concurrent execution.

By default, BEAM uses as many schedulers as there are CPU cores available. For example, on a quad-core machine, four schedulers are used, as shown in figure 5.1.

**Figure 5.1. BEAM as a single OS process, using a few threads to schedule a large number of processes**




Each scheduler runs in its own thread, and the entire VM runs in a single OS process. In the previous example, you have one OS process and four OS threads, and that's all you need to run a highly concurrent server system.

A scheduler is in charge of the interchangeable execution of processes. Each process gets an execution time slot; after the time is up, the

running process is preempted, and the next one takes over.

Processes are light. It takes only a couple of microseconds to create a single process, and its initial memory footprint is 1 to 2 KB. By comparison, OS threads usually use a couple of megabytes just for the stack. Therefore, you can create a large number of processes: the theoretical limit imposed by the VM is roughly 268 million!

This feature can be exploited in server-side systems to manage various tasks that should run simultaneously. Using a dedicated process for each task, you can take advantage of all available CPU cores and parallelize the work as much as possible.

Moreover, running tasks in different processes improves the server's reliability and fault tolerance. BEAM processes are completely isolated: they share no memory, and a crash of one process won't take down other processes. In addition, BEAM provides a means to detect a process crash and do something about it: for example, restarting the crashed process. All this makes it easier to create systems that are more stable and can gracefully recover from unexpected errors, which inevitably occur in production.

Finally, each process can manage some state and can receive messages from other processes to manipulate or retrieve that state. As you saw in part 1 of this book, data in Elixir is immutable. To keep it alive, you have to hold on to it, constantly passing the result of some function to another one. A process can be considered a container of this data—a place where an immutable structure is stored and kept alive for a longer time, possibly forever.

As you see, there's more to concurrency than parallelization of the work. With this high-level view of BEAM processes in place, let's see how you can create processes and work with them.

## 5.2. Working with processes

The benefits of processes are most obvious when you want to run

something concurrently and parallelize the work as much as possible. For example, let's say you need to run a bunch of potentially long-running database queries. You can run those queries sequentially, one at a time, or you can try to run them concurrently, hoping that the total execution time will speed up.

**Concurrency vs. parallelism**

It's important to realize that concurrency doesn't necessary imply parallelism. Two concurrent *things* have independent execution contexts. But this doesn't mean they will run in parallel. If you run two CPU-bound concurrent tasks and only one CPU core, then parallel execution can't happen.

Of course, by adding more CPU cores and relying on an efficient concurrent framework, you can achieve parallelism. But you should be aware that concurrency itself doesn't necessarily speed things up.

To keep things simple, you'll use a simulation of a long-running database query, presented in a following snippet:

```
iex(1)> run_query = fn(query_def) ->
          :timer.sleep(2000)
          "#{query_def} result"
        end
```

Simulates a long running time

Here, the code sleeps for 2 seconds to simulate a long-running operation. When you call the `run_query` lambda, the shell is blocked until the lambda is done:

```
iex(2)> run_query.("query 1")

"query 1 result"
```

2 seconds later

Consequently, if you run 5 queries, it will take 10 seconds to get all the results:

```
iex(3)> 1..5 |>
        Enum.map(&run_query.("query #{&1}"))

["query 1 result", "query 2 result", "query 3 result"
 "query 4 result", "query 5 result"]
```

10 seconds later

Obviously, this is neither performant nor scalable. Assuming that the queries are already optimized, the only thing you can do to try to make things faster is to run the queries concurrently. This won't speed up the running time of a single query, but the total time required to run all the queries should be much less. In the BEAM world, to run something concurrently, you have to create a separate process.

### 5.2.1. Creating processes

To create a process, you can use the auto-imported `spawn/1` function:

```
spawn(fn ->
  expression_1
  ...
  expression_n
end)
```

Runs in the new process

The function `spawn/1` takes a zero-arity lambda that will run in the new process. After the process is created, `spawn` immediately returns, and the caller process's execution continues. The provided lambda is executed in the new process and therefore runs concurrently. After the lambda is done, the spawned process exits, and its memory is released.

You can try this to run the query concurrently:

```
iex(4)> spawn(fn -> IO.puts(run_query.("query 1")) end)
#PID<0.48.0>                                          <-- Immediately returned

result of query 1                  <-------------- Printed after 2 seconds
```

As you can see, the call to `spawn/1` returns immediately, and you can do something else in the shell while the query runs. Then, after 2 seconds, the result is printed to the screen. This happens because you called `IO.puts/1` from a separate process.

The funny-looking `#PID<0.48.0>` that is returned by `spawn/1` is the identifier of the created process, often called a *pid*. This can be used to communicate with the process, as you'll see later in this

chapter.

In the meantime, let's play some more with concurrent execution. First, you'll create a helper lambda that concurrently runs the query and prints the result:

```
iex(5)> async_query = fn(query_def) ->
          spawn(fn -> IO.puts(run_query.(query_def)) end)
        end

iex(6)> async_query.("query 1")
#PID<0.52.0>

result of query 1          <-- 2 seconds later
```

This code demonstrates an important technique: how to pass data to the created process. Notice that `async_query` takes one argument and binds it to the `query_def` variable. This data is then passed to the newly created process via the closure mechanism. The inner lambda—the one that runs in a separate process—references the variable `query_def` from the outer scope. This results in cross-process data passing—the contents of `query_def` are passed from the main process to the newly created one. When it's passed to another process, the data is deep-copied, because two processes can't share any memory.

**Note**

In BEAM, everything runs in a process. This also holds for the interactive shell. All expressions you enter in `iex` are executed in a single, shell-specific process. In this example, the main process is the shell process.

Now that you have the `async_query` lambda in place, you can try to run five queries concurrently:

```
iex(7)> Enum.each(1..5, &async_query.("query #{&1}"))
:ok                                          <-- Returns immediately

result of query 5
result of query 4
result of query 3          After 2 seconds
result of query 2
result of query 1
```

As expected, the call to `Enum.each/2` now returns immediately (in the first, sequential version you had to wait 10 seconds for it to finish). Moreover, all the results are printed at practically the same time, 2 seconds later, which is a five-fold improvement over the sequential version! This happens because you run each computation concurrently.

Also note that, because processes run concurrently, the order of execution isn't guaranteed.

In contrast to the sequential version, the caller process doesn't get the result of the spawned processes. The processes run concurrently, each one printing the result to the screen. At the same time, the caller process runs independently and has no access to any data from the spawned processes. Remember, processes are completely independent and isolated.

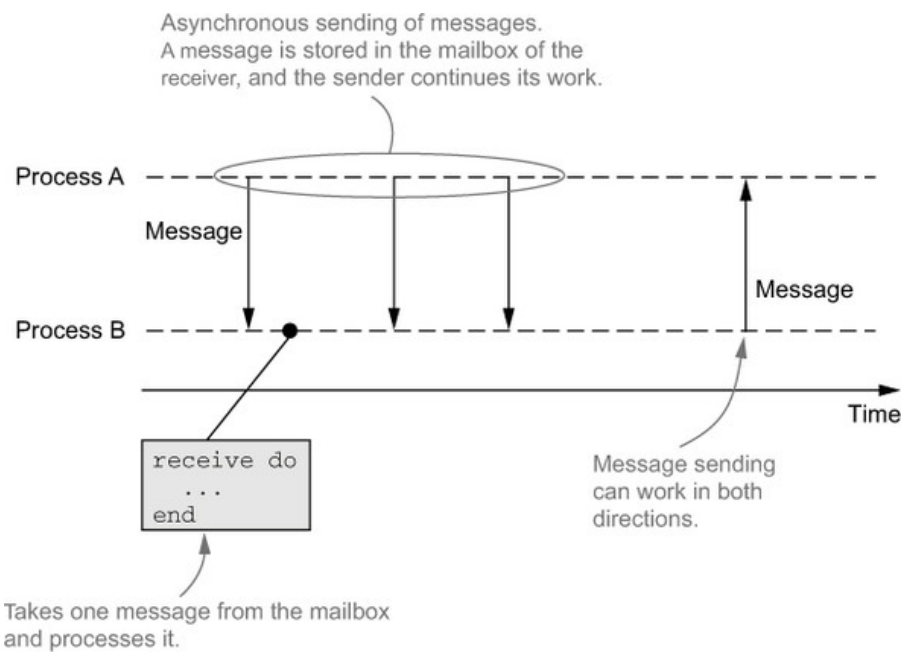
Many times, a simple "fire and forget" concurrent execution, where the caller process doesn't receive any notification from the spawned ones, will suffice. Sometimes you want to return the result of the concurrent operation to the caller process. For this purpose, you can use the message-passing mechanism.

### 5.2.2. Message passing

In complex systems, you often need concurrent tasks to cooperate in some way. For example, you may have a main process that spawns multiple concurrent calculations, and then you want to handle all the results in the main process.

Being completely isolated, processes can't use shared data structures to exchange knowledge. Instead, processes communicate via messages, as illustrated in figure 5.2.

**Figure 5.2. Inter-process communication via messages**




When process A wants process B to do something, it sends an asynchronous message to B. The content of the message is an Elixir term—anything you can store in a variable. Sending a message amounts to storing it into the receiver's mailbox. The caller then continues with its own execution, and the receiver can pull the message in any time and process it in some way. Because processes can't share memory, a message is deep-copied when it's being sent.

The process mailbox is a FIFO queue limited only by the available memory. Therefore, the receiver consumes messages in the order received, and a message can be removed from the queue only if it's consumed.

To send a message to a process, you need to have access to its process identifier (pid). Recall from the previous section that the pid of the newly created process is the result of the `spawn/1` function. In addition, you can obtain the pid of the current process by calling the auto-imported `self/1` function.

Once you have a receiver's pid, you can send it messages using the `Kernel.send/2` function:

```
send(pid, {:an, :arbitrary, :term})
```

The consequence of `send` is that a message is placed in the mailbox of the receiver. The caller then continues doing something else.

On the receiver side, to pull a message from the mailbox, you have to use the `receive` construct:

```
receive do
  pattern_1 -> do_something
  pattern_2 -> do_something_else
end
```

The `receive` construct works similarly to the `case` expression you saw in chapter 3. It tries to pull one message from the process mailbox, match it against any of the provided patterns, and run the corresponding code. You can easily test this by making the shell process send messages to itself:

```
iex(1)> send(self, "a message")
iex(2)> receive do
          message -> IO.inspect message
        end
"a message"
```

Sends the message

Receives the message

If you want to handle a specific message, you can rely on pattern matching:

```
iex(3)> send(self, {:message, 1})
iex(4)> receive do
          {:message, id} ->
            IO.puts "received message #{id}"
        end
received message 1
```

Pattern-matches the message

If there are no messages in the mailbox, `receive` waits indefinitely for a new message to arrive. The following call blocks the shell, and you need to manually terminate it:

```
iex(5)> receive do
          message -> IO.inspect message
        end
```

The shell is blocked because the process mailbox is empty.

The same thing happens if a message can't be matched against

provided pattern clauses:

```
iex(1)> send(self, {:message, 1})

iex(2)> receive do
          {_, _, _} ->
            IO.puts "received"
        end
```

This doesn't match the sent message ...

... so the shell is blocked

If you don't want `receive` to block, you can specify the `after` clause that is executed if the message wasn't received in a given time frame (in milliseconds):

```
iex(1)> receive do
          message -> IO.inspect message
          after 5000 -> IO.puts "message not received"
        end

message not received
```

5 seconds later

**Receive algorithm**

Recall from chapter 3 that an error is raised when you can't pattern-match the given term. The `receive` construct is an exception to this rule. If a message doesn't match any of the provided clauses, it's put back into the process mailbox, and the next message is processed.

The `receive` construct works as follows:

1. Take the first message from the mailbox.
2. Try to match it against any of the provided patterns, going from top to bottom.
3. If a pattern matches the message, run the corresponding code.
4. If no pattern matches, put the message back into the mailbox at the same position it originally occupied. Then try the next message.
5. If there are no more messages in the queue, wait for a new one to arrive. When a new message arrives, start from step 1,

inspecting the first message in the mailbox.

6. If the `after` clause is specified and no message arrives in the given amount of time, run the code from the `after` block.

As you already know, each Elixir expression returns a value, and `receive` is no exception. The result of `receive` is the result of the last expression in the appropriate clause:

```
iex(1)> send(self, {:message, 1})

iex(2)> receive_result = receive do
          {:message, x} ->
            x + 2                <-- The result of receive
        end

iex(3)> IO.inspect receive_result
3
```

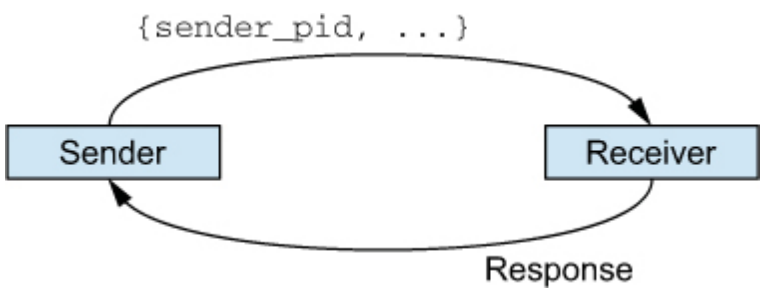
To summarize, `receive` tries to find the first (oldest) message in the process mailbox that can be matched against any of the provided clauses. If such a message is found, the corresponding code is executed. Otherwise, `receive` waits for such a message for a specified amount of time, or indefinitely if the `after` clause isn't provided.

#### Synchronous sending

The basic message-passing mechanism is the asynchronous, "fire and forget" kind. A process sends a message and then continues to run, oblivious of what happens in the receiver. Sometimes a caller needs some kind of response from the receiver. There is no special language construct for doing this. Instead, you must program both parties to cooperate using the basic asynchronous messaging facility.

The caller must include its own pid in the message contents and then wait for a response from the receiver. The receiver uses the embedded pid to send the response to the caller, as illustrated in figure 5.3. You'll see this in action a bit later, when we discuss server processes.

**Figure 5.3. Synchronous send and receive implemented on top of the asynchronous messages**




**Collecting query results**

Let's try message-passing with the concurrent queries developed in the previous section. In your initial attempt, you'll run queries in separate processes and print them to the screen from those processes. Then you'll try to collect all the results in the main process.

First let's recall how the `async_query` lambda works:

```
iex(1)> run_query = fn(query_def) ->
          :timer.sleep(2000)
          "#{query_def} result"
        end
iex(2)> async_query = fn(query_def) ->
          spawn(fn -> IO.puts(run_query.(query_def)) end)
        end
```

Instead of printing to the screen, let's make the lambda send the query result to the caller process:

```
iex(3)> async_query = fn(query_def) ->
          caller = self                    <-- Stores the pid of the calling process

          spawn(fn ->
            send(caller, {:query_result, run_query.(query_def)})    <-- Responds to the calling process
          end)
        end
```

In this code, you first store the pid of the calling process to a distinct `caller` variable. This is necessary so the worker process (the one doing the calculation) can know the pid of the process that receives the response.

Keep in mind that the result of `self/0` depends on the calling process. If you didn't store the result to the `caller` variable, and you tried to `send(self, ...)` from the inner lambda, it would have no

effect. The spawned process would send the message to itself, because calling `self` from within a process returns the pid of that process.

The worker process can now use the `caller` variable to return the result of the calculation. The message is in the custom format `{:query_result, result}`. This makes it possible to distinguish between your messages and any others that might be sent to the caller process.

Now you can start your queries:

```
iex(4)> Enum.each(1..5, &async_query.("query #{&1}"))
```

This runs all the queries concurrently, and the result is stored in the mailbox of the caller process. In this case, this is the shell (`iex`) process.

Notice that the caller process is neither blocked nor interrupted while receiving messages. Sending a message doesn't disturb the receiving process in any way. If the process is performing computations, it continues to do so. The only thing affected is the content of the receiving process's mailbox. Messages remain in the mailbox until they're consumed or the process terminates.

Let's get the results. First you make a lambda that pulls one message from the mailbox and extracts the query result from it:

```
iex(5)> get_result = fn ->
          receive do
            {:query_result, result} -> result
          end
        end
```

Now you can pull all the messages from the mailbox into a single list:

```
iex(6)> results = Enum.map(1..5, fn(_) -> get_result.() end)

["query 3 result", "query 2 result", "query 1 result",
  "query 5 result", "query 4 result"]
```

Notice the use of `Enum.map/2`, which maps anything enumerable to

a list of the same length. In this case, you create a range of size 5 and then map each element to the result of the `get_result` lambda. Of course, this works because you know there are five messages waiting for you. Otherwise, the loop would get stuck waiting for new messages to arrive.

It's also worth pointing out that results arrive in nondeterministic order. Because all computations run concurrently, it's not certain in which order they will finish.

This is a simple implementation of a parallel map technique that can be used to process a larger amount of work in parallel and then collect the results into a list. Let's see the complete version once more:

```
iex(7)> 1..5                                          |>
        Enum.map(&async_query.("query #{&1}"))        |>
        Enum.map(fn(_) -> get_result.() end)
```

Starts concurrent computations

Collects the results

This concludes the basics of process creation and communication. In addition to pure work parallelization, there is another important use case for processes: maintaining a mutable state.

## 5.3. Stateful server processes

Spawning processes to perform one-off tasks isn't the only use case for concurrency. In Elixir, it's common to create long-running processes that can respond to various messages. Such processes can keep their internal state, which other processes can query or even manipulate.

In this sense, stateful server processes resemble objects. They maintain state and can interact with other processes via messages. But a process is concurrent, so multiple server processes may run in parallel.

Server processes are an important concept in Elixir/Erlang systems, so we'll spend some time exploring this topic.

### 5.3.1. Server processes

A *server process* is an informal name for a process that runs for a long time (or forever) and can handle various requests (messages). To make a process run forever, you have to use endless tail recursion. You may remember from chapter 3 that tail calls receive special treatment. If the last thing a function does is call another function (or itself), then a simple jump takes place instead of a stack push. Consequently, a function that always calls itself will run forever, without causing a stack overflow or consuming additional memory.

This can be used to implement a server process. You need to run the endless loop and wait for a message in each step of the loop. When the message is received, you handle it and then continue the loop. Let's try this and turn the query example into a server process. The basic sketch is provided in the following listing.

**Listing 5.1. Long-running server process that runs queries (database_server.ex)**

```
defmodule DatabaseServer do
  def start do
    spawn(&loop/0)              <-- Starts the loop concurrently
  end

  defp loop do
    receive do
      ...                       |  Handles one message
    end

    loop                        <-- Keeps the loop running
  end

  ...
end
```

`start/0` is the so-called *interface function* that is used by clients to start the server process. When `start/0` is called, it creates the long-running process that runs forever. This is ensured in the private `loop/0` function, which waits for a message, handles it, and finally calls itself, thus ensuring that the process never stops. This loop usually isn't CPU intensive. Waiting for a message puts the process in a suspended state and doesn't waste CPU cycles.

Notice that functions in this module run in different processes. The function `start/0` is called by clients and runs in a client process. The private function `loop/0` runs in the server process. It's perfectly normal to have different functions from the same module running in different processes—there is no special relationship between modules and processes. A module is just a collection of functions, and these functions can be invoked in any process.

When implementing a server process, it usually makes sense to put all of its code in a single module. The functions of this module generally fall in two categories: interface and implementation. *Interface functions* are public and are executed in the caller process. They hide the details of process creation and the communication protocol. *Implementation functions* are usually private and run in the server process.

**Note**

As was the case with classical loops, you typically won't need to code the recursion loop yourself. A standard abstraction called `gen_server` (generic server) is provided, which simplifies the development of stateful server processes. The abstraction still relies on recursion, but this recursion is implemented in `gen_server`. You'll learn about this abstraction in chapter 7.

Next, let's see the full implementation of the `loop/0` function.

**Listing 5.2. Database server loop (database_server.ex)**

```
defmodule DatabaseServer do
  ...
  defp loop do
    receive do                                    <-- Awaits a message
      {:run_query, caller, query_def} ->
        send(caller, {:query_result, run_query(query_def)})   <-- Runs the query and sends the response to the caller
    end

    loop
  end

  defp run_query(query_def) do
    :timer.sleep(2000)                            <-- Query execution
    "#{query_def} result"
  end

  ...
end
```

This code reveals the communication protocol between the caller

process and the database server. The caller sends a message in the format `{:run_query, caller, number}`. The server process handles such a message by executing the query and sending the query result back to the caller process.

Usually you want to hide these communication details from your clients. Clients shouldn't depend on knowing the exact structure of messages that must be sent or received. To hide this, it's best to provide a dedicated interface function. Let's introduce a function called `run_async/2` that will be used by clients to request the operation—in this case, a query execution—from the server. This function makes the clients unaware of message-passing details—they just call `run_async/2` and get the result. The implementation is given in the following listing.

**Listing 5.3. Implementation of `run_async/2` (database_server.ex)**

```
defmodule DatabaseServer do
  ...

  def run_async(server_pid, query_def) do
    send(server_pid, {:run_query, self, query_def})
  end

  ...
end
```

The `run_async/2` function receives the pid of the database server and a query you want to execute. It sends the appropriate message to the server and then does nothing else. Calling `run_async/2` from the client requests that the server process run the query while the caller goes about its business.

Once the query is executed, the server sends a message to the caller process. To get this result, you need to add another interface function. It's called `get_result/0`, and its implementation is given next.

**Listing 5.4. Implementation of `get_result/0` (database_server.ex)**

```
defmodule DatabaseServer do
  ...

  def get_result do
    receive do
      {:query_result, result} -> result
    after 5000 ->
      {:error, :timeout}
    end
```

```
  end

  ...
end
```

`get_result/0` is called when the client wants to get the query result. Here, you use `receive` to get the message. The `after` clause ensures that you give up after some time passes—for example, if something goes wrong during the query execution and a response never comes back.

The database server is now complete. Let's see how to use it:

```
iex(1)> server_pid = DatabaseServer.start

iex(2)> DatabaseServer.run_async(server_pid, "query 1")
iex(3)> DatabaseServer.get_result
"query 1 result"

iex(3)> DatabaseServer.run_async(server_pid, "query 2")

iex(4)> DatabaseServer.get_result
"query 2 result"
```

Notice how you execute multiple queries on the same process. First you run query 1 and then query 2. This proves that the server process continues running after a message is received.

Because communication details are wrapped in functions, the client isn't aware of them. Instead, it communicates with the process with plain functions. Here, the server pid plays an important role. You receive the pid by calling `DatabaseServer.start/0`, and then you use it to issue requests to the server.

Of course, the request is handled asynchronously in the server process. After calling `DatabaseServer.run_async/2`, you can do whatever you want in the client (`iex`) process and collect the result when you need it.

#### Server processes are sequential

It's important to realize that a server process is internally sequential. It runs a loop that processes one message at a time. Thus, if you issue

five asynchronous query requests to a single server process, they will be handled one by one, and the result of the last query will come after 10 seconds.

This is a good thing, because it helps you reason about the system. A server process can be considered a synchronization point. If multiple actions need to happen synchronously, in a serialized manner, you can introduce a single process and forward all requests to that process, which handles the requests sequentially.

Of course, in this case, a sequential property is a problem. You want to run multiple queries concurrently, to get the result as quickly as possible. What can you do about it?

Assuming that the queries can be run independently, you can start a pool of server processes, and then for each query somehow choose one of the processes from the pool and have that process run the query. If the pool is large enough, and you divide the work uniformly across each worker in the pool, you'll parallelize the total work as much as possible.

Here is a basic sketch of how can this be done. First, let's create a pool of database-server processes:

```
iex(1)> pool =
          1..100 |>
            Enum.map(fn(_) -> DatabaseServer.start end)
```

Here you create 100 database-server processes and store their pids in a list. You may think that 100 processes is a lot, but recall that processes are lightweight. They take up a small amount of memory (~2 KB) and are created very quickly (a few microseconds). Furthermore, because all of these processes wait for a message, they're effectively idle and don't waste CPU time.

Next, when you run a query, you need to decide which process will execute the query. The simplest way is to use the `:random.uniform/1` function, which takes a positive integer and returns a random number in the range 1..*N* (inclusive). Taking advantage of this, the following expression distributes five queries

over a pool of processes:

Note that this isn't efficient. You're using `Enum.at/2` to get the pid at a random position. Because you use a list to keep the processes, and a random lookup is an O(N) operation, selecting a random worker isn't very performant. You could do better if you used a `HashDict` structure with process indexes as keys and pids as values; and there are other alternatives (such as using a round-robin approach). But for now, let's stick with this simple implementation.

Once you've queued the queries to the workers, you need to collect the responses. This is now straightforward, as illustrated in the following snippet:

```
iex(3)> 1..5 |>
          Enum.map(fn(_) -> DatabaseServer.get_result end)

["5 result", "3 result", "1 result", "4 result", "2 result"]
```

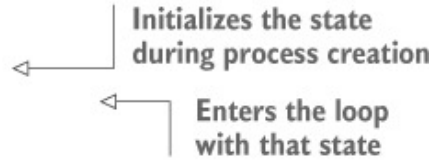
Thanks to this, you get all the results much faster, because queries are again executed concurrently.

### 5.3.2. Keeping a process state

Server processes open the possibility of keeping some kind of process-specific state. For example, when you talk to a database, you need to have a connection handle that is used to communicate with the server. If your process is responsible for TCP communication, it needs to keep the corresponding socket.

To keep a state in the process, you can extend the `loop` function with additional argument(s). Here is a basic sketch:

```
def start do
  spawn(fn ->
    initial_state = ...
    loop(initial_state)
  end)
end

defp loop(state) do
  ...
  loop(state)
end
```




Let's use this technique to extend your database server with a connection. In this example, you'll use a random number as a simulation of the connection handle. First you need to initialize the connection while the process starts, as demonstrated in the following listing.

**Listing 5.5. Initializing the process state (stateful_database_server.ex)**

```
defmodule DatabaseServer do
  ...

  def start do
    spawn(fn ->
      connection = :random.uniform(1000)
      loop(connection)
    end)
  end

  ...
end
```

Here, you open the connection and then pass the corresponding handle to the `loop` function. In real life, instead of generating a random number, you'd use a database client library (such as ODBC) to open the connection.

Next, you need to modify the `loop` function.

**Listing 5.6. Using the connection while querying (stateful_database_server.ex)**

```
defmodule DatabaseServer do
  ...

  defp loop(connection) do
    receive do
      {:run_query, from_pid, query_def} ->
        query_result = run_query(connection, query_def)
        send(from_pid, {:query_result, query_result})
    end

    loop(connection)
  end

  defp run_query(connection, query_def) do
    :timer.sleep(2000)
    "Connection #{connection}: #{query_def} result"
  end

  ...
end
```

Uses the connection while running the query

Keeps the connection in the loop argument

This is fairly straightforward. The `loop` function keeps the state (connection) as the first argument. You have to additionally extend the `run_query` function to use the connection while querying the database. The connection handle (in this case, a number) is included in the query result.

With this, your stateful database server is complete. Notice that you didn't change the interface of its public functions, so the usage remains the same as it was. Let's see how it works:

```
iex(1)> server_pid = DatabaseServer.start

iex(2)> DatabaseServer.run_async(server_pid, "query 1")
iex(3)> DatabaseServer.get_result
"Connection 753: query 1 result"

iex(4)> DatabaseServer.run_async(server_pid, "query 2")
iex(5)> DatabaseServer.get_result
"Connection 753: query 2 result"
```

The results for different queries are executed using the same connection handle, which is kept internally in the process loop and is completely invisible to other processes.

### 5.3.3. Mutable state

So far, you've seen how to keep a constant, process-specific state. It doesn't take much to make this state mutable. Here is the basic idea:

```
def loop(state) do
  new_state = receive do
    msg1 ->
      ...

    msg2 ->
      ...
  end

  loop(new_state)
end
```

Computes the new state based on the received message

Loops with the new state

This is a typical stateful server technique. You wait for a message and then, based on its contents, compute the new state. Finally, you loop recursively with the new state, thus effectively changing the state. The next received message operates on the new state.

From the outside, stateful processes are mutable. By sending messages to a process, a caller can affect its state and the outcome of subsequent requests handled in that server. In that sense, sending a message is an operation with possible side effects. Still, the server relies on immutable data structures. A state can be any valid Elixir variable ranging from simple numbers to complex data abstractions, such as `TodoList` (which you saw in chapter 4).

Let's see this in action. You'll start with a simple example: a stateful calculator process that keeps a number as its state. Initially, the state of the process is 0, and you can manipulate it by issuing requests such as `add`, `sub`, `mul`, and `div`. You can also retrieve the process state with the `value` request.

Here's how you use the server:

```
iex(1)> calculator_pid = Calculator.start

iex(2)> Calculator.value(calculator_pid)
0

iex(3)> Calculator.add(calculator_pid, 10)
iex(4)> Calculator.sub(calculator_pid, 5)
iex(5)> Calculator.mul(calculator_pid, 3)
iex(6)> Calculator.div(calculator_pid, 5)

iex(7)> Calculator.value(calculator_pid)
3.0
```

Starts the process

Verifies the initial value

Issues requests

Verifies the value

In this code, you start the process and check its initial state. Then you issue some modifier requests and verify the result of the operations `(((0 + 10) - 5) * 3) / 5`, which is `3.0`.

Now it's time to implement this. First, let's look at the server's inner loop.

**Listing 5.7. Concurrent stateful calculator (calculator.ex)**

```
defmodule Calculator do
  ...

  defp loop(current_value) do
    new_value = receive do
      {:value, caller} ->
        send(caller, {:response, current_value})
        current_value

      {:add, value} -> current_value + value
      {:sub, value} -> current_value - value
      {:mul, value} -> current_value * value
      {:div, value} -> current_value / value

      invalid_request ->
        IO.puts "invalid request #{inspect invalid_request}"
        current_value
    end

    loop(new_value)
  end

  ...
end
```

Getter request

Arithmetic operations requests

Unsupported request

The loop handles various messages. The `value` message is used to retrieve the server's state. Because you need to send the response back, the caller must include its pid in the message. Notice that the last expression of this block returns `current_value`. This is needed because the result of `receive` is stored in `new_value`, which is then used as the server's new state. By returning `current_value`, you specify that the `value` request doesn't change the process state.

The arithmetic operations compute the new state based on the current value and the argument received in the message. Unlike a `value` message handler, arithmetic operation handlers don't send responses back to the caller. This makes it possible to run these operations asynchronously, as you'll see soon when you implement interface functions.

The final `receive` clause matches all the other messages. These are the ones you're not supporting, so you log them to the screen and return `current_value`, leaving the state unchanged.

Next, you have to implement the interface functions that will be used by clients. This is done in the next listing.

**Listing 5.8. Calculator interface functions (calculator.ex)**

```
defmodule Calculator do
  def start do
    spawn(fn -> loop(0) end)
  end

  def value(server_pid) do
    send(server_pid, {:value, self})
    receive do
      {:response, value} ->
        value
    end
  end

  def add(server_pid, value), do: send(server_pid, {:add, value})
  def sub(server_pid, value), do: send(server_pid, {:sub, value})
  def mul(server_pid, value), do: send(server_pid, {:mul, value})
  def div(server_pid, value), do: send(server_pid, {:div, value})

  ...
end
```

Starts the server, and initializes the state

The value request

Arithmetic operations

The interface functions are straightforward and follow the protocol specified in the `loop/1` function. The `value` request is an example of the synchronous message passing mentioned in section 5.2.2. There is nothing special happening here—you send a message and immediately wait for the response. Notice that this waiting blocks the caller until the response comes back, thus making the request-handling synchronous.

The arithmetic operations run asynchronously. There is no response message, so the caller doesn't have to wait for anything. Therefore, a caller can issue a number of these requests and continue doing its own work while the operations run concurrently in the server process. Keep in mind that the server handles messages in the order received, so requests are handled in the proper order.

Why do you make the arithmetic operations asynchronous? Because you don't care when they're executed. Until you request the server's state (via the `value/1` function), you don't want the client to block.

This makes your code more concurrent, because the client doesn't block while the server is doing a computation. Moreover, it minimizes possible deadlock situations, because the client process isn't blocked while the server is processing a request.

#### Refactoring the loop

As you introduce multiple requests to your server, the `loop` function becomes more complex. If you have to handle many requests, it will become bloated, turning into a huge `switch/case`-like construct. You can refactor this by relying on pattern matching and moving the message handling to a separate multiclause function. This keeps the code of the `loop` function very simple:

```
defp loop(current_value) do
  new_value = receive do
    message ->
      process_message(current_value, message)
  end

  loop(new_value)
end
```

Looking at this code, you can see the general workflow of the server. A message is first received and then processed. Message processing generally amounts to computing the new state based on the current state and the message received. Finally, you loop with this new state, effectively setting it in place of the old one.

`process_message` is a simple multiclause function that receives the current state and the message. Its task is to perform message-specific code and return the new state:

```
defp process_message(current_value, {:value, caller}) do
  send(caller, {:response, current_value})
  current_value
end

defp process_message(current_value, {:add, value}) do
  current_value + value
end

...
```

This code is a simple reorganization of the server process loop. It allows you to keep the loop code compact and to move the message-

handling details to a separate multiclause function, with each clause handling a specific message.

### 5.3.4. Complex states

Usually a state is much more complex than a simple number. But the technique remains the same—you keep the mutable state using the private `loop` function. As the state becomes more complex, the code of the server process can become increasingly complicated. It's worth extracting the state manipulation to a separate module and keeping the server process focused only on passing messages and keeping the state.

Let's look at this technique using the `TodoList` abstraction developed in chapter 4. First, let's recall the basic usage of the structure:

```
iex(1)> todo_list = TodoList.new |>
          TodoList.add_entry(
              %{date: {2013, 12, 19}, title: "Dentist"}
          ) |>

          TodoList.add_entry(
            %{date: {2013, 12, 20}, title: "Shopping"}
          ) |>

          TodoList.add_entry(
            %{date: {2013, 12, 19}, title: "Movies"}
          )

iex(2)> TodoList.entries(todo_list, {2013, 12, 19})

[%{date: {2013, 12, 19}, id: 3, title: "Movies"},
 %{date: {2013, 12, 19}, id: 1, title: "Dentist"}]

[TodoList.Entry[id: 1, date: {2013, 12, 19}, title: "Dentist"],
 TodoList.Entry[id: 3, date: {2013, 12, 19}, title: "Movies"]]
```

As you may recall, a `TodoList` is a pure functional abstraction. To keep the structure alive, you constantly must hold on to the result of the last operation performed on the structure.

In this example, you'll build a `TodoServer` module that keeps this abstraction in the private state. Let's see how the server is used:

```
iex(1)> todo_server = TodoServer.start

iex(2)> TodoServer.add_entry(todo_server,
          %{date: {2013, 12, 19}, title: "Dentist"})

iex(3)> TodoServer.add_entry(todo_server,
          %{date: {2013, 12, 20}, title: "Shopping"})

iex(4)> TodoServer.add_entry(todo_server,
          %{date: {2013, 12, 19}, title: "Movies"})

iex(5)> TodoServer.entries(todo_server, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 3, title: "Movies"},
 %{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

You start the server and then use its pid to manipulate the data. In contrast to the pure functional approach, you don't need to take the result of a modification and feed it as an argument to the next operation. Instead, you constantly use the same `todo_server` variable to manipulate the to-do list.

In a sense, `todo_server` works more like an object reference or a pointer in a classical OO language. Whatever you do on a server is preserved as long as the server is running. Of course, unlike objects, the server operations are running concurrently to the caller.

Let's start implementing this server. First you need to place all the modules in a single file, as shown in the following listing.

**Listing 5.9. `TodoServer` modules (todo_server.ex)**

```
defmodule TodoServer do
  ...
end

defmodule TodoList do
  ...
end
```

Putting both modules in the same file ensures that you have everything available when you load the file while starting the `iex` shell. Of course, in more complicated systems, you'll use the proper `mix` project, as explained in chapter 7; but for now, this is sufficient.

The `TodoList` implementation is the same as in chapter 4. It has all the features you need to use it in a server process.

Let's set up the basic structure of the to-do server process, as shown next.

**Listing 5.10. `TodoServer` basic structure (todo_server.ex)**

```
defmodule TodoServer do
  def start do
    spawn(fn -> loop(TodoList.new) end)
  end

  defp loop(todo_list) do
    new_todo_list = receive do
      message ->
        process_message(todo_list, message)
    end

    loop(new_todo_list)
  end

  ...
end
```

Uses a to-do list as the initial state

There is nothing new here. You start the loop using a new instance of the `TodoList` abstraction as the initial state. In the loop, you receive messages and apply them to the state by calling the `process_message/2` function, which returns the new state. Finally, you loop with the new state.

For each request you want to support, you have to add a dedicated clause in the `process_message/2` function. Additionally, a corresponding interface function must be introduced. Let's begin by supporting the `add_entry` request.

**Listing 5.11. The `add_entry` request (todo_server.ex)**

```
defmodule TodoServer do
  ...

  def add_entry(todo_server, new_entry) do
    send(todo_server, {:add_entry, new_entry})
  end

  ...

  defp process_message(todo_list, {:add_entry, new_entry}) do
    TodoList.add_entry(todo_list, new_entry)
  end

  ...
end
```

Interface function

Message-handler clause

The interface function sends the new entry data to the server. Recall that the `loop` function calls `process_message/2`, and the call

ends up in the `process_message/2` clause. Here, you delegate to the `TodoList` function and return the modified `TodoList` instance. This returned instance is used as the new server's state.

Using a similar approach, you can implement the `entries` request, keeping in mind that you need to wait for the response message. The code is shown in the next listing.

**Listing 5.12. The `entries` request (todo_server.ex)**

```
defmodule TodoServer do
  ...
  def entries(todo_server, date) do
    send(todo_server, {:entries, self, date})

    receive do
      {:todo_entries, entries} -> entries
    after 5000 ->
      {:error, :timeout}
    end
  end

  ...

  defp process_message(todo_list, {:entries, caller, date}) do
    send(caller, {:todo_entries, TodoList.entries(todo_list, date)})
    todo_list
  end

  ...
end
```

Sends the response to the caller

The state remains unchanged.

Again, this is a synthesis of techniques you've seen previously. You send a message and wait for the response. In the corresponding `process_message/2` clause, you delegate to `TodoList`, and then you send the response and return the unchanged to-do list. This is needed because `loop/2` takes the result of `process_message/2` as the new state.

In a similar way, you can add support for other to-do list requests such as `update_entry` and `delete_entry`. The implementation of these requests is left for you as an exercise.

**Concurrent vs. functional approach**

A process that keeps a mutable state can be regarded as a a kind of mutable data structure. You can start the server and then perform various requests on it. But you shouldn't abuse this technique versus the pure functional approach.

The role of a stateful process is to keep the data available while the system is running. The data should be modeled using pure functional abstractions, just as you did with `TodoList`. A pure functional structure provides many benefits, such as integrity and atomicity. Furthermore, it can be reused in various contexts and tested independently.

A stateful process is then used on top of functional abstractions as a kind of concurrent controller that keeps the state and can be used to manipulate that state or read parts of it. For example, if you're implementing a web server that manages multiple to-do lists, you most likely have one server process for each to-do list. While handling an HTTP request, you can find the corresponding to-do server and have it run the requested operation. Each to-do list manipulation runs concurrently, thus making your server scalable and more performant. Moreover, there are no synchronization problems, because each to-do list is managed in a dedicated process. Recall that a single process is always sequential. So multiple competing requests that manipulate the same to-do list are serialized and handled sequentially in the corresponding process. Don't worry if this seems vague—you'll see it in action in next chapter.

### 5.3.5. Registered processes

In order for a process to cooperate with other processes, it must know their whereabouts. In BEAM, a process is identified by the corresponding pid. To make process A send messages to process B, you have to bring the pid of process B to process A. In this sense, a pid resembles a reference or pointer in the OO world.

Sometimes it can be cumbersome to keep and pass pids. If you know there will always be only one instance of some type of server, you can *register* it under a *local alias*. Then you can use that alias to send messages to the process. The alias is called *local* because it has meaning only in the currently running BEAM instance. This distinction becomes important when you start connecting multiple BEAM instances to a distributed system, as you'll see in chapter 12.

Registering a process can be done with `Process.register(pid,`

`alias)` where `alias` must be an atom. Here's a quick illustration:

```
iex(1)> Process.register(self, :some_name)          <-- Registers a process
iex(2)> send(:some_name, :msg)          <-- Sends a message via a symbolic name
iex(3)> receive do
          msg -> IO.puts "received #{msg}"
        end                                  Verifies that the message is received
received msg
```

The following rules apply to registered processes:

- The process alias can only be an atom.
- A single process can have only one alias.
- Two processes can't have the same alias.

If these rules aren't satisfied, an error is raised. In addition, trying to send a message to a nonexistent alias also results in an error.

For practice, try to change the to-do server to run as a registered process. The interface of the server will then be simplified, because you don't need to keep and pass the server's pid. Here's an example of how such a server can be used:

```
iex(1)> TodoServer.start

iex(2)> TodoServer.add_entry(%{date: {2013, 12, 19}, title:
"Dentist"})
iex(3)> TodoServer.add_entry(%{date: {2013, 12, 20}, title:
"Shopping"})
iex(4)> TodoServer.add_entry(%{date: {2013, 12, 19}, title:
"Movies"})

iex(5)> TodoServer.entries({2013, 12, 19})
[%{date: {2013, 12, 19}, id: 3, title: "Movies"},
 %{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

To make this work, you have to register a server process under a symbolic alias (such as `:todo_server`). Then, you change all the interface functions to use the registered name when sending a message to the process. If you get stuck, the solution is provided in the registered_todo_server.ex file.

The usage of the registered server is much simpler because you don't

have to store the server's pid and pass it to the interface functions. Instead, the interface functions internally use the registered alias to send messages to the process.

Local registration plays an important role in process discovery. Registered aliases provide a way of communicating with a process without knowing its pid. This becomes increasingly important when you start dealing with restarting processes (as you'll see in chapters 8 and 9) and distributed systems (described in chapter 12).

With this in mind, we've finished with our initial exploration of stateful processes. They play an important role in Elixir-based systems, and you'll continue using them throughout the book. But at this point, we'll look at some important runtime properties of BEAM processes.

## 5.4. Runtime considerations

By now, you've learned a great deal about how processes work. To take full advantage of BEAM concurrency, it's important to understand some of its internals. Don't worry—we won't go too deep and will only deal with relevant topics.

### 5.4.1. A process is a bottleneck

It has already been mentioned, but it's so important, we'll stress it again: although multiple processes may run in parallel, a single process is always sequential—it either runs some code or waits for a message. If many processes send messages to a single process, then that single process can significantly affect overall throughput.

Let's see an example. The code in the following listing implements a slow echo server.

**Listing 5.13. Demonstration of a process bottleneck (process_bottleneck.ex)**

```
defmodule Server do
  def start do
    spawn(fn -> loop end)
  end

  def send_msg(server, message) do
    send(server, {self, message})
    receive do
      {:response, response} -> response
    end
  end

  defp loop do
    receive do
      {caller, msg} ->
        :timer.sleep(1000)
        send(caller, {:response, msg})
    end

    loop
  end
end
```

Simulates long processing

Echoes the message back

Upon receiving a message, the server sends the message back to the caller. Before that, it sleeps for a second to simulate a long-running request.

Let's fire up five concurrent clients:

```
iex(1)> server = Server.start

iex(2)> Enum.each(1..5, fn(i) ->
          spawn(fn ->
            IO.puts "Sending msg ##{i}"
            response = Server.send_msg(server, i)
            IO.puts "Response: #{response}"
          end)
        end)
```

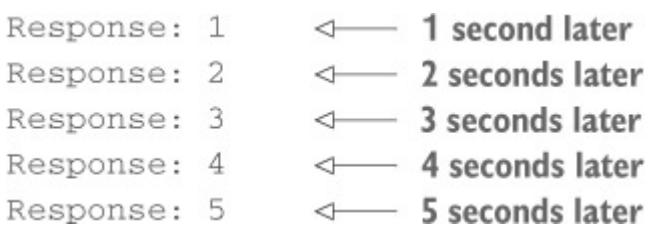
Spawns a concurrent client

Sends a message to the server

As soon as you start this, you'll see the following lines printed:

```
Sending msg #1
Sending msg #2
Sending msg #3
Sending msg #4
Sending msg #5
```

So far, so good. Five processes have been started and are running concurrently. But now the problems begin—the responses come back slowly, one by one, a second apart:




What happened? The echo server can handle only one message per second. Because all other processes depend on the echo server, they're constrained by its throughput.

What can you do about this? Once you identify the bottleneck, you should try to optimize the process internally. Generally, a server process has a simple flow. It receives and handles message one by one. So the goal is to make the server handle messages at least as fast as they arrive. In this example, server optimization amounts to removing the `:timer.sleep` call.

If you can't make message handling fast enough, you can try to split the server into multiple processes, effectively parallelizing the original work and hoping that doing so will boost performance on a multicore system. This should be your last resort, though. Parallelization isn't a remedy for a poorly structured algorithm.

### 5.4.2. Unlimited process mailboxes

Theoretically, a process mailbox has an unlimited size. In practice, the mailbox size is limited by available memory. Thus, if a process constantly falls behind, meaning messages arrive faster than the process can handle them, the mailbox will constantly grow and increasingly consume memory. Ultimately, a single slow process may cause an entire system to crash by consuming all the available memory.

A more subtle version of the same problem occurs if a process doesn't handle some messages at all. Consider the following server loop:

```
def loop
  receive
```

```
    {:message, msg} ->
      do_something(msg)
  end

  loop
end
```

A server powered by this loop handles only messages that are in the form `{:message, something}`. All other messages remain in the process mailbox forever, taking up memory space for no reason.

In addition, large mailbox contents cause performance slowdowns. Recall how pattern matching works in the `receive` construct: messages are analyzed one by one, from oldest to newest, until a message is matched. Let's say your process contains a million unhandled messages. When a new message arrives, `receive` iterates through the existing million messages before processing the new one. Consequently, the process wastes time iterating through unhandled messages, and its throughput suffers.

How can you resolve this problem? For each server process, you should introduce a *match all* receive clause that deals with unexpected kinds of messages. Typically, you'll log that a process has received the unknown message, and do nothing else about it:

```
def loop
  receive
    {:message, msg} ->
      do_something(msg)

    other ->                    <-- Match-all clause
      log_unknown_message(other)
  end

  loop
end
```

The server in this case handles all types of messages, logging the unexpected ones.

It's also worth noting that BEAM gives us tools to analyze processes in

the runtime. In particular, you can query each process for its mailbox size and thus detect the ones for which the mailbox-queue buildup occurs. We'll discuss this feature in chapter 13.

### 5.4.3. Shared nothing concurrency

As already mentioned, processes share no memory. Thus, sending a message to another process results in a deep copy of the message contents:

```
send(target_pid, data)          <-- The data variable is deep copied.
```

Less obviously, closing on a variable from a spawn also results in deep copying of the closed variable:

```
data = ...

spawn(fn ->
  ...
  some_fun(data)          <-- Results in a deep copy of the data variable
  ...
end)
```

This is something you should be aware of when moving code into a separate process. Deep copying is an in-memory operation, so it should be reasonably fast. Occasionally sending a big message shouldn't present a problem. But having many processes frequently send big messages may affect the performance of the system. The notion of small versus big is subjective. Simple data, such as a number, an atom, or a tuple with few elements, is obviously small. A list of a million complex structures is big. The border lies somewhere in between and depends on your specific case.

A special case where deep copying doesn't take place involves binaries (including strings) that are larger than 64 bytes. These are maintained on a special shared binary heap, and sending them doesn't result in a deep copy. This can be useful when you need to send information to many processes, and the processes don't need to decode the string.

You may wonder about the purpose of shared nothing concurrency. First, it simplifies the code of each individual process. Because processes don't share memory, you don't need complicated synchronization mechanisms such as locks and mutexes. Another benefit is overall stability: one process can't compromise the memory of another. This in turn promotes the integrity and fault-tolerance of the system. Finally, shared nothing concurrency makes it possible to implement an efficient garbage collector.

Because processes share no memory, garbage collection can take place on a process level. Each process gets an initial small chunk of heap memory (~2 KB on 64-bit BEAM). When more memory is needed, garbage collection for that process takes place and is included in the process execution window.

So instead of one large "stop the entire system" collection, you have many smaller collections. This in turn keeps the entire system more responsive, without larger, unexpected blockages during garbage collections. It's possible for one scheduler to perform a micro-collection while the others are doing meaningful work.

### 5.4.4. Scheduler inner workings

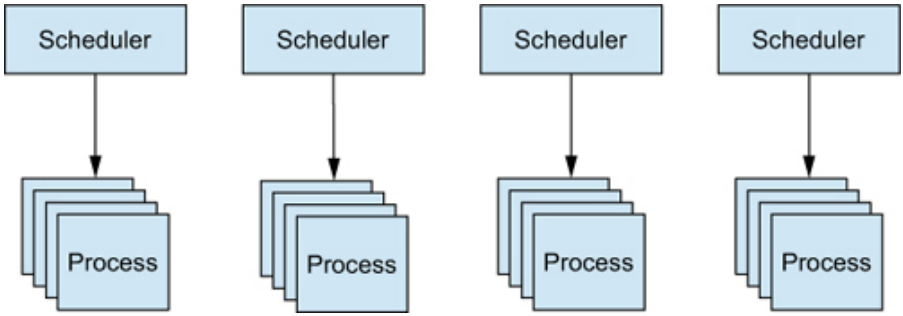
Each BEAM scheduler is in reality an OS thread that manages the execution of BEAM processes. By default, BEAM uses only as many schedulers as there are logical processors available. You can change these settings via various Erlang emulator flags. To provide those flags, you can use the following syntax:

```
$ iex --erl "put Erlang emulator flags here"
```

A list of all Erlang flags can be found at http://erlang.org/doc/man/erl.html.

In general, you can assume that there are *n* schedulers that run *m* processes, with *m* most often being significantly larger than *n*. This is called *m:n* threading, and it reflects the fact that you run a large number of logical microthreads using a smaller number of OS threads, as illustrated in figure 5.4.

**Figure 5.4. *m:n* threading: a small number of schedulers running a large number of BEAM processes**




Internally, each scheduler maintains a *run queue*, which is something like a list of BEAM processes it's responsible for. Each process gets a small execution window, after which it's preempted and another process is executed. The execution window is approximately 2,000 function calls (internally called *reductions*). Because you're dealing with a functional language where functions are very small, it's clear that context switching happens often, generally in less than 1 millisecond.

This promotes the responsiveness of BEAM powered systems. If one process performs a long CPU-bound operation, such as computing the value of pi to a billion decimals, it won't block the entire scheduler, and other processes shouldn't be affected.

There are some special cases when a process will implicitly yield execution to the scheduler before its execution time is up. The most notable situation is when using `receive`. Another example is a call to the `:timer.sleep/1` function. In both cases, the process is suspended, and the scheduler can run other processes.

Another important case of implicit yielding involves I/O operations, which are internally executed on separate threads called *async threads*. When issuing an I/O call, the calling process is preempted, and other processes get the execution slot. After the I/O operation finishes, the scheduler resumes the calling process. A great benefit of this is that your I/O code looks synchronous, while under the hood it still runs asynchronously. By default, BEAM fires up 10 async threads, but you can change this via the `+A n` Erlang flag.

Additionally, if your OS supports it, you can rely on a kernel poll such as *epoll* or *kqueue*, which takes advantage of the OS kernel for nonblocking I/O. You can request the use of a kernel poll by providing the `+K true` Erlang flag when you start the BEAM.

Implicit yields provide additional benefits. If most processes are suspended most of the time—for example, while the kernel is doing I/O or while many processes are waiting for messages—BEAM schedulers are even more efficient and have bigger overall throughput.

## 5.5. Summary

This concludes our initial tour of concurrency, which plays a central role in the development of highly available systems. You've learned about the basic mechanisms of creating BEAM processes and working with them. The important takeaways of this chapter are as follows:

- A BEAM process is a lightweight concurrent unit of execution. Processes are completely isolated and share no memory.
- Processes can communicate with asynchronous messages. Synchronous sends and responses are manually built on top of this basic mechanism.
- A server process is a process that runs for a long time (possibly forever) and handles various messages. Server processes are powered by endless recursion.
- Server processes can maintain their own private state using the arguments of the endless recursion.

At this point you're aware of the basic mechanisms in BEAM concurrency. But we need to study some of its aspect in more detail. In the next chapter, we'll take a closer look at generic server processes.

# Chapter 6. Generic server processes

*This chapter covers*

- Building a generic server process
- Using `gen_server`

In chapter 5, you became familiar with basic concurrency techniques: you learned how to create processes and communicate with them. I also explained the idea behind stateful server processes—long-running processes that maintain a state and can react to messages, do some processing, optionally send a response, and maybe change the internal process state.

Server processes play an important role and are used frequently when building highly concurrent systems in Elixir and Erlang, so we'll spend some time exploring them in detail. In this chapter, you'll learn how to reduce some of the boilerplate associated with server processes, such as infinite recursion, state management, and message passing.

Erlang provides a helper for implementing server processes as the part of the framework called Open Telecom Platform (OTP). Despite its misleading name, the framework has nothing to do with telecoms; rather, it provides patterns and abstractions for tasks such as creating components, building releases, developing server processes, handling and recovering from runtime errors, logging, event handling, and upgrading code.

You'll learn about various parts of OTP throughout this book, but in this chapter we'll focus on one of its most important part: `gen_server`—a helper that simplifies implementation of server processes. Before we look at `gen_server`, you'll implement a simplified version of it based on message-passing primitives you saw in chapter 5.

## 6.1. Building a generic server process

You saw a few examples of server processes in chapter 5. Although those processes serve different purposes, there are some commonalities in their implementations. In particular, all code that implements a server process needs to do the following tasks:

- Spawn a separate process.
- Run an infinite loop in the process.
- Maintain the process state.
- React to messages.
- Send a response back to the caller.

No matter what kind of a server process you run, you'll always need to do these tasks, so it's worth moving this code to a single place. Concrete implementations can then reuse this code and focus on their specific needs. Let's see how to implement such generic code.

### 6.1.1. Plugging in with modules

The generic code will perform various tasks common to server processes, leaving the specific decisions to concrete implementations. For example, the generic code will spawn a process, but the concrete implementation must determine the initial state. Similarly, the generic code will receive a message and optionally send the response, but the concrete implementation must decide how the message is handled and what the response is.

In other words, the generic code drives the entire process, and the specific implementation must fill in the missing pieces. Therefore, you need a plug-in mechanism that lets the generic code call into the concrete implementation when a specific decision needs to be made.

The simplest way to do this is to use modules. Remember that a module name is an atom. You can store that atom into a variable and use the variable later to invoke functions on it:

```
iex(1)> some_module = IO
iex(2)> some_module.puts("Hello")
Hello
```

Stores a module atom into a variable

Dynamic invocation

You can use this feature to provide callback hooks from the generic code. In particular, you can take the following approach:

1. Make the generic code accept a plug-in module as the argument. That module is called a *callback module*.
2. Maintain the module atom as part of the process state.
3. Invoke callback-module functions when needed.

Obviously, for this to work, a callback module must implement and export a well-defined set of functions, which we'll gradually introduce as we implement the generic code.

### 6.1.2. Implementing the generic code

Let's start building a generic server process. First you need to start the process and initialize its state, as shown in the following listing.

**Listing 6.1. Starting the server process (server_process.ex)**

```
defmodule ServerProcess do
  def start(callback_module) do
    spawn(fn ->
      initial_state = callback_module.init
      loop(callback_module, initial_state)
    end)
  end

  ...
end
```

Invokes the callback to initialize the state

`ServerProcess.start/1` takes a module atom as the argument and then spawns the process. In the spawned process, the callback function `init/0` is invoked to create the initial state. Obviously, for this to work, the callback module must export the `init/0` function.

Finally, you enter the loop that will power the server process and maintain this state. The return value of `ServerProcess.start/1` is a pid, which can be used to send messages to the request process.

Next, you need to implement the loop code that powers the process, waits for messages, and handles them. In this example, you'll implement a synchronous send-and-response communication pattern. The server process must receive a message, handle it, send the response message back to the caller, and change the process state.

The generic code is responsible for receiving and sending messages, whereas the specific implementation must handle the message and return the response and the new state. The idea is illustrated in the following listing.

**Listing 6.2. Handling messages in the server process (server_process.ex)**

```
defmodule ServerProcess do
  ...
  defp loop(callback_module, current_state) do
    receive do
      {request, caller} ->
        {response, new_state} = callback_module.handle_call(
          request,
          current_state
        )

        send(caller, {:response, response})

        loop(callback_module, new_state)
    end
  end

  ...
end
```

Invokes the callback to handle the message

Sends the response back

Loops with the new state

Here, you expect a message in the form of a `{request, caller}` tuple. The `request` is data that identifies the request and is meaningful to the specific implementation. The callback function `handle_call/2` takes the request payload and the current state, and it must return a `{response, new_state}` tuple. The generic code can then send the response back to the caller and continue looping with the new state.

There's only one thing left to do: you need to provide a function to issue requests to the server process.

**Listing 6.3. Helper for issuing requests (server_process.ex)**

```
defmodule ServerProcess do
  ...
  def call(server_pid, request) do
    send(server_pid, {request, self})          <-- Sends the message
    receive do                                 <-- Waits for the response
      {:response, response} ->
        response                               <-- Returns the response
    end
  end
end
```

At this point you have the abstraction for the generic server process in place. Let's see how it can be used.

### 6.1.3. Using the generic abstraction

To test the server process, you'll implement a simple key-value store. It will be a process that can be used to store mappings between arbitrary terms.

Remember that the callback module must implement two functions: `init/0`, which creates the initial state, and `handle_call/2`, which handles specific requests. The code is shown next.

**Listing 6.4. Key-value store implementation (server_process.ex)**

```
defmodule KeyValueStore do
  def init do
    HashDict.new                    <-- Initial process state
  end
  def handle_call({:put, key, value}, state) do
    {:ok, HashDict.put(state, key, value)}
   end                                         Handles the put request

  def handle_call({:get, key}, state) do
    {HashDict.get(state, key), state}          Handles the get request
  end
end
```

That's all it takes to create a specific server process. Because the infinite loop and message-passing boilerplate are pushed to the generic code, the specific implementation is more concise and focused on its main task.

Take particular notice how you use a multiclause in `handle_call/2` to handle different types of requests. This is the place where the specific implementation decides how to handle each request. The `ServerProcess` module is generic code that blindly forwards requests from client processes to the callback module.

Let's test the process:

```
iex(1)> pid = ServerProcess.start(KeyValueStore)

iex(2)> ServerProcess.call(pid, {:put, :some_key, :some_value})
:ok

iex(3)> ServerProcess.call(pid, {:get, :some_key})
:some_value
```

Notice how you start the process with `ServerProcess.start(KeyValueStore)`. This is where you plug the specific `KeyValueStore` into the generic code of `ServerProcess`. All subsequent invocations of `ServerProcess.call/2` will send messages to that process, which will in turn call `KeyValueStore.handle_call/2` to perform the handling.

It's beneficial to make clients completely oblivious to the fact that the `Server-Process` abstraction is used. This can be achieved by introducing helper functions, as shown here.

**Listing 6.5. Wrapping `ServerProcess` function calls (server_process.ex)**

```
defmodule KeyValueStore do
  def start do
    ServerProcess.start(KeyValueStore)
  end

  def put(pid, key, value) do
    ServerProcess.call(pid, {:put, key, value})
  end

  def get(pid, key) do
    ServerProcess.call(pid, {:get, key})
  end

  ...
end
```

Clients can now use `start/0`, `put/3`, and `get/2` to manipulate the

key-value store. These functions are informally called *interface functions*—clients use the API of `KeyValueStore` to start and interact with the process.

In contrast, `init/0` and `handle_call/2` are callback functions used internally by the generic code. Note that interface functions run in client processes, whereas callback functions are always invoked in the server process.

### 6.1.4. Supporting asynchronous requests

The current implementation of `ServerProcess` supports only synchronous requests. Let's expand on this and introduce support for asynchronous fire-and-forget requests, where a client sends a message and doesn't wait for a response.

In the current code, we use the term *call* for synchronous requests. For asynchronous requests, we'll use the term *cast*. This is the naming convention used in OTP, so it's good to adapt to it.

Because you're introducing the second request type, you need to change the format of messages that are passed between client processes and the server. This will allow you to determine the request type in the server process and handle different types of requests in different ways.

This can be as simple as including the request-type information in the tuple being passed from the client process to the server, as shown next.

**Listing 6.6. Including the request type in the message (server_process_cast.ex)**

```
defmodule ServerProcess do
  ...
  def call(server_pid, request) do
    send(server_pid, {:call, request, self()})
    ...
  end
  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->
        ...
    end
  end
  ...
end
```

Tags the request message as a call

Handles a call request

Now you can introduce support for `cast` requests. In this scenario, when the message arrives, the specific implementation handles it and returns the new state. No response is sent back to the caller, so the callback function must return only the new state. The code is provided in the following listing.

**Listing 6.7. Supporting casts in the server process (server_process_cast.ex)**

```
defmodule ServerProcess do
  ...
  def cast(server_pid, request) do
    send(server_pid, {:cast, request})
  end
  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->
        ...
      {:cast, request} ->
        new_state = callback_module.handle_cast(
          request,
          current_state
        )
        loop(callback_module, new_state)
    end
  end
  ...
end
```

Issues a cast message

Handles a cast message

To handle a cast request, you need the callback function `handle_cast/2`. This function must handle the message and return

the new state. In the server loop, you then invoke this function and loop with the new state. That's all it takes to support cast requests.

Finally, let's change the implementation of the key-value store to use casts. Keep in mind that a cast is a fire-and-forget type of request, so it's not suitable for all requests. In this example, the `get` request must be a call, because the server process needs to respond with the value associated with a given key. In contrast, the `put` request can be implemented as a cast because the client doesn't need to wait for the response.

**Listing 6.8. Implementing `put` as a cast (server_process_cast.ex)**

```
defmodule KeyValueStore do
  ...

  def put(pid, key, value) do
    ServerProcess.cast(pid, {:put, key, value})
  end

  ...

  def handle_cast({:put, key, value}, state) do
    HashDict.put(state, key, value)
  end

  ...
end
```

Issues the put request as a cast

Handles the put request

Now you can try the server process:

```
iex(1)> pid = KeyValueStore.start

iex(2)> KeyValueStore.put(pid, :some_key, :some_value)

iex(3)> KeyValueStore.get(pid, :some_key)
:some_value
```

With a simple change in the generic implementation, you added another feature to the service processes. Specific implementations can now decide whether each concrete request should be implemented as a call or as a cast.

### 6.1.5. Exercise: refactoring the to-do server

An important benefit of the generic `ServerProcess` abstraction is that it lets you easily create various kinds of processes that rely on the

common code. For example, in chapter 5, you developed a simple to-do server that maintains a to-do list abstraction in its internal state. This server can also be powered by the generic `ServerProcess`.

This is the perfect opportunity for you to practice a bit. Take the complete code from todo_server.ex from the chapter 5 source, and save it to a different file. Then add the last version of the `ServerProcess` module to the same file. Finally, adapt the code of the `TodoServer` module to work with `ServerProcess`.

Once you have everything working, compare the code between the two versions. The new version of `TodoServer` should be smaller and simpler, even for such a simple server process that supports only two different requests. If you get stuck, you can find the solution in the server_process_todo.ex file.

**Note**

It is of course clumsy to place multiple modules in a single file and maintain multiple copies of the `ServerProcess` code in different files. In chapter 7, you'll start using a better approach to code organization powered by the `mix` tool. But for the moment, let's stick with the current simplistic approach.

In the meantime, you're finished implementing a basic abstraction for generic server processes. The current implementation is simple and leaves a lot of room for improvement, but it demonstrates the basic technique of generic server processes. Next it's time to use the full-blown OTP abstraction for generic server processes: `gen_server`.

## 6.2. Using gen_server

When it comes to production-ready code, it doesn't make much sense to build and use the manually baked `ServerProcess` abstraction. The reason is that OTP ships with a much better support for generic server processes, called `gen_server`. In addition to being much more feature rich than `ServerProcess`, `gen_server` also handles various kinds of edge cases and is battle-tested in production in

complex concurrent systems.

Some of the compelling features provided by `gen_server` include the following:

- Support for calls and casts
- Customizable timeouts for call requests
- Propagation of server-process crashes to client processes waiting for a response
- Support for distributed systems

Note that there's no special magic behind `gen_server`. Its code relies on concurrency primitives explained in chapter 5 and fault-tolerance features explained in chapter 9. After all, `gen_server` is implemented in plain Erlang, and (with a lot of work) you could reimplement it yourself in Elixir.

In this section, you'll learn how to build your server processes with `gen_server`. But first, let's examine the concept of OTP *behaviours*.

**Note**

Notice the British spelling of the word *behaviour*: this is the preferred spelling used both in code and in official documentation. This book uses the British spelling to specifically denote an OTP behaviour but retains the American spelling (*behavior*) for all other purposes.

### 6.2.1. OTP behaviours

`ServerProcess` is a simple example of a behaviour. In Erlang terminology, a behaviour is generic code that implements a common pattern. The generic logic is exposed through the behaviour module, and you can plug into it by implementing a corresponding callback module. The callback module must satisfy a contract defined by the behaviour, meaning it must implement and export a set of functions. The behaviour module then calls into these functions, thus allowing you to provide your own specialization of the generic code.

This is exactly what `ServerProcess` does. It powers a generic server process, requiring specific implementations to provide the callback module that implements `init/0`, `handle_call/2` and `handle_cast/2` functions.

It's even possible to specify the behaviour contract and verify that the callback module implements required functions during compilation. For details, see the official documentation for the `Behaviour` module (http://mng.bz/Yu1Q).

OTP ships with a few predefined behaviours:

- `gen_server`—Generic implementation of a stateful server process
- `supervisor`—Provides error handling and recovery in concurrent systems
- `application`—Generic implementation of components and libraries
- `gen_event`—Provides event-handling support
- `gen_fsm`—Runs a finite state machine in a stateful server process

This book focuses on the first three behaviours. The `gen_server` behaviour receives detailed treatment in this chapter and chapter 7, `supervisor` is discussed in chapters 8 and 9, and `application` is presented in chapter 11.

The remaining two behaviours, `gen_event` and `gen_fsm`, although useful, are used less often and won't be discussed in this book. Once you get a grip on `gen_server` and `supervisor`, you should be able to research other behaviours on your own time and use them when the need arises.

### 6.2.2. Plugging into gen_server

Using `gen_server` is roughly similar to using `ServerProcess`. There are some differences in the format of the returned values, but the basic idea is the same.

In total, the `gen_server` behaviour requires six callback functions, but frequently you'll need only a subset of those. To simplify the implementation, you can reach for Elixir's `GenServer` module. You can get some sensible default implementations of all required callback functions if you *use* the `GenServer` module:

```
iex(1)> defmodule KeyValueStore do
          use GenServer
        end
```

The `use` macro is a language construct you haven't seen previously. During compilation, when this instruction is encountered, the specific macro from the `GenServer` module is invoked. That macro then injects a bunch of functions into the calling module (`KeyValueStore`, in this case). You can verify this in the shell:

```
iex(2)> KeyValueStore.__info__(:functions)
[code_change: 3, handle_call: 3, handle_cast: 2, handle_info: 2,
 init: 1, terminate: 2]
```

Here you use the `__info__/1` function that is automatically injected into each Elixir module during compilation. It lists all exported functions of a module (except `__info__/1`).

As you can see in the output, many functions are automatically included in the module due to `use GenServer`. These are all callback functions that need to be implemented for you to plug into the `gen_server` behaviour.

Of course, you can then override the default implementation of each function as required. If you define the function of the same name and arity in your module, it will overwrite the default implementation you get through `use`.

At this point, you can plug your callback module into the behaviour. To start the process, use the `GenServer.start/2` function:

```
iex(3)> GenServer.start(KeyValueStore, nil)
{:ok, #PID<0.51.0>}
```

This works roughly like `ServerProcess`. The server process is

started, and the behaviour uses `KeyValueStore` as the callback module. The second argument of `GenServer.start/2` is a custom parameter that is passed to the process during its initialization. For the moment, you don't need this, so you send the `nil` value. Finally, notice that the result of `GenServer.start/2` is a tuple of the form `{:ok, pid}`.

### 6.2.3. Handling requests

Now you can convert the `KeyValueStore` to work with `gen_server`. To do this, you need to implement three callbacks: `init/1`, `handle_cast/2`, and `handle_call/3`. These callbacks work similarly to the ones in `ServerProcess`, with a couple of differences:

- `init/1` accepts one argument. This is the second argument provided to `GenServer.start/2`, and you can use it to pass data to the server process while starting it.
- The result of `init/1` must be in the format `{:ok, initial_state}`. Alternatively, you can return `{:stop, some_reason}` if for some reason you decide the server process shouldn't be started.
- `handle_cast/2` accepts the request and the state and should return the result in the format `{:noreply, new_state}`.
- `handle_call/3` takes the request, the caller information, and the state. It should return the result in the format `{:reply, response, new_state}`.

With those differences in mind, the following listing implements the callback functions.

**Listing 6.9. Implementing `gen_server` callbacks (key_value_gen_server.ex)**

```
defmodule KeyValueStore do
  use GenServer

  def init(_) do
    {:ok, HashDict.new}
  end

  def handle_cast({:put, key, value}, state) do
    {:noreply, HashDict.put(state, key, value)}
  end
```

```
  def handle_call({:get, key}, _, state) do
    {:reply, HashDict.get(state, key), state}
  end
end
```

The second argument to `handle_call/3` is a tuple that contains the request ID (used internally by the `gen_server` behaviour) and the pid of the caller. This information is in most cases not needed, so in this example you ignore it.

With these callbacks in place, the only things missing are interface functions. To interact with a `gen_server` process, you can use functions from the `GenServer` module. In particular, you can use `GenServer.start/2` to start the process and `GenServer.cast/2` and `GenServer.call/2` to issue requests. The implementation is then straightforward.

**Listing 6.10. Adding interface functions (key_value_gen_server.ex)**

```
defmodule KeyValueStore do
  use GenServer

  def start do
    GenServer.start(KeyValueStore, nil)
  end

  def put(pid, key, value) do
    GenServer.cast(pid, {:put, key, value})
  end

  def get(pid, key) do
    GenServer.call(pid, {:get, key})
  end

  ...
end
```

That's it! With only a few changes, you've moved from a basic `ServerProcess` to a full-blown `gen_server`. Let's test the server:

```
iex(1)> {:ok, pid} = KeyValueStore.start

iex(2)> KeyValueStore.put(pid, :some_key, :some_value)

iex(3)> KeyValueStore.get(pid, :some_key)
:some_value
```

It works as expected.

There are many differences between `ServerProcess` and `gen_server`, but a few points deserve special mention. The result of `GenServer.start/2` is a tuple `{:ok, pid}`. Alternatively, if in `init/1` you decide against starting the server and return `{:stop, reason}`, then the result of `start/2` is `{:error, reason}`.

You should also be aware that `GenServer.start/2` works synchronously. In other words, `start/2` returns only after the `init/1` callback has finished in the server process. Consequently, the client process that starts the server is blocked until the server process is initialized.

Finally, note that `GenServer.call/2` doesn't wait indefinitely for a response. By default, if the response message doesn't arrive in five seconds, an error is raised in the client process. You can alter this by using `GenServer.call(pid, request, timeout)`, where `timeout` is given in milliseconds. In addition, if the receiver process happens to terminate while you're waiting for the response, `gen_server` detects it and raises a corresponding error in the caller process.

### 6.2.4. Handling plain messages

Messages sent to the server process via `GenServer.call` and `GenServer.cast` contain more than just a request payload. Those functions include additional data in the message sent to the server process. This is something you did in the `ServerProcess` example in section 6.1:

```
defmodule ServerProcess do
  ...
  def call(server_pid, request) do
    send(server_pid, {:call, request, self})    <-- Calls a message
    ...
  end
  def cast(server_pid, request) do
    send(server_pid, {:cast, request})    <-- Casts a message
  end

  ...
  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->          <-- Special handling of a call message
        ...
      {:cast, request} ->          <-- Special handling of a cast message
        ...
    end
  end

  ...
end
```

Notice that you don't send the plain `request` payload to the server process; you include additional data, such as the request type and the caller for call requests.

`gen_server` uses a similar approach, using `:"$gen_cast"` and `:"$gen_call"` atoms to decorate cast and call messages. You don't need to worry about the exact format of those messages, but it's important to understand that `gen_server` internally uses particular message formats and handles those messages in a specific way.

Occasionally you may need to handle messages that aren't specific to `gen_server`. For example, imagine that you need to do a periodic cleanup of the server process state. You can use the Erlang function `:timer.send_interval/2`, which periodically sends a message to the caller process. Because this message isn't a `gen_server`-specific message, it's not treated as a cast or a call. Instead, for such plain messages, `gen_server` calls the `handle_info/2` callback, giving you a chance to do something with the message.

Here's the sketch of this technique:

```
iex(1)> defmodule KeyValueStore do
          use GenServer

          def init(_) do
            :timer.send_interval(5000, :cleanup)
            {:ok, HashDict.new}
          end

          def handle_info(:cleanup, state) do
            IO.puts "performing cleanup..."
            {:noreply, state}
          end

          def handle_info(_, state), do: {:noreply, state}
        end

iex(2)> GenServer.start(KeyValueStore, nil)

performing cleanup...
performing cleanup...
performing cleanup...
```

Sets up periodic message sending

Handles the plain :cleanup message

Handles all other messages

Printed every five seconds

During process initialization, you make sure a `:cleanup message` is sent to the process every five seconds. This message is handled in the `handle_info/2` callback, which essentially works like `handle_cast/2`, returning the result as `{:noreply, new_state}`.

Notice how another match-all clause is defined for `handle_info/2` and handles all other plain messages. Such an implementation is by default provided via `use GenServer`. But as soon as you redefine a function, the default implementation is overwritten, and you have to provide this additional clause.

Without this match-all clause, any other message sent to your process would crash the server process. Remember, the callback function is invoked by the `gen_server` behaviour. If your callback function doesn't match the provided arguments, an error is raised. Such errors aren't handled by `gen_server`, and, consequently, the server process will crash.

Notice that you didn't use match-all clauses for `handle_cast` and `handle_call` in previous examples. This was intentional. Casts and calls are well-defined requests. They specify an interface between clients and the server process. Thus an invalid cast means your clients are using an unsupported interface. In such cases, you usually want to fail fast and signal an error.

On the other hand, plain messages are something you don't have control over. A process may occasionally receive a VM-specific message even if you didn't ask for it. Thus you should always provide a default `handle_info/2` clause that does nothing. But if your callback module doesn't define `handle_info/2`, you don't have to do this—it's included as a default implementation, courtesy of `use GenServer`.

### 6.2.5. Other gen_server features

There are various other features and subtleties not mentioned in this basic introduction to `gen_server`. You'll learn about some of them throughout the book, but you should definitely take the time to research the documentation for the `GenServer` module (http://mng.bz/7ZZd) and its Erlang counterpart (http://mng.bz/40B8). A couple of points still deserve special mention.

#### Alias registration

Recall from chapter 5 that a process can be registered under a local alias (an atom), where *local* means the alias is registered only in the currently running BEAM instance. This allows you to create a singleton process that you can access by name without needing to know its pid.

Local registration is an important feature because it supports patterns of fault-tolerance and distributed systems. You'll see exactly how this works in later chapters, but it's worth mentioning that you can provide the process alias as an option to `GenServer.start`:

```
GenServer.start(
  CallbackModule,
  init_param,
  name: :some_alias
)
```

Registers the process under an alias

You can then issue calls/casts using the alias:

```
GenServer.call(:some_alias, ...)
GenServer.cast(:some_alias, ...)
```

Different callbacks can return various types of responses. So far, you've seen the most common cases:

- `{:ok, initial_state}` from `init/1`
- `{:reply, response, new_state}` from `handle_call/3`
- `{:noreply, new_state}` from `handle_cast/2` and `handle_info/2`

There are additional possibilities, the most important one being the option to stop the server process.

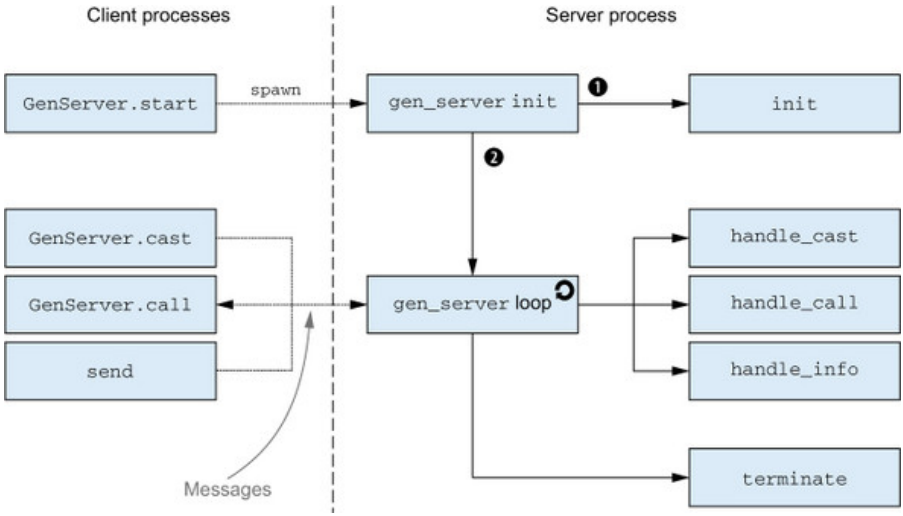
Returning `{:stop, reason, new_state}` from `handle_*` callbacks causes `gen_ server` to stop the server process. If the termination is part of the standard workflow, you should use the atom `:normal` as the stoppage reason. If you're in `handle_call/3` and also need to respond to the caller before terminating, you can return `{:stop, reason, response, new_state}`.

You may wonder why you need to return a new state if you're terminating the process. The reason is that just before the termination, `gen_server` calls the callback function `terminate/2`, sending it the termination reason and the final state of the process. This can be useful if you need to perform cleanup.

### 6.2.6. Process life cycle

It's important to always be aware of how `gen_server`-powered processes tick and where (in which process) various functions are executed. Let's do a quick recap by looking at figure 6.1, which shows the life cycle of a typical server process.

**Figure 6.1. Life cycle of a `gen_server`-powered process**




A client process starts the server by calling `GenServer.start` and providing the callback module (1). This creates the new server process, which is powered by the `gen_server` behaviour.

Requests can be issued by client processes using various `GenServer` functions or plain `send`. When a message is received, `gen_server` invokes callback functions to handle it. Therefore, callback functions are always executed in the server process.

The process state is maintained in the `gen_server` loop but is defined and manipulated by the callback functions. It starts with `init/1`, which defines the initial state that is then passed to subsequent `handle_*` callbacks (2). Each of these callbacks receives the current state and must return its new version, which is used by the `gen_server` loop in place of the old one.

**The Actor model**

Erlang is an accidental implementation of the *Actor model* originally described by Carl Hewitt. An *actor* is a concurrent computational entity that encapsulates state and can communicate with other actors. When processing a single message, an actor can designate the new state that will be used when processing the next message. This is

roughly similar to how `gen_server`-based processes work in Erlang. Note, though, that as Robert Virding (one of Erlang's co-inventors) has repeatedly stated, Erlang developers arrived at this idea on their own and learned about the existence of the Actor model much later.

There are some disagreements about whether Erlang is a proper implementation of the Actor model, and the term *actor* isn't used much in the Erlang community. This book doesn't use this terminology either. Still, it's worth keeping in mind that in the context of Erlang, an actor corresponds to a server process, most frequently a `gen_server`.

### 6.2.7. OTP-compliant processes

For various reasons, once you start building production systems, you should usually avoid using plain processes started with `spawn`. Instead, all of your processes should be so-called *OTP-compliant processes*. Such processes adhere to OTP conventions, they can be used in supervision trees (described in chapter 9), and errors in those processes are logged with more details.

All processes powered by OTP behaviours such as `gen_server` and `supervisor` are OTP compliant. In addition, Elixir comes with higher-level abstractions that implement OTP-compliant processes. In Elixir 1.0, two such abstractions are available: *tasks* and *agents*. Tasks (http://mng.bz/Dok1) make it simple to start a concurrent job and collect the result later. Agents (http://mng.bz/B297) are a simpler (but less powerful) alternative to `gen_server`-based processes and are appropriate if the single purpose of the process is to manage and expose state. Although these abstractions can make your code simpler, this book won't deal with them. The reason is that they're built on top of OTP concepts such as `gen_server` and `supervisor` (which you'll meet in chapter 8). Once you get a grasp on how OTP behaviours work, it will be simple to understand abstractions such as tasks and agents.

### 6.2.8. Exercise: gen_server-powered to-do server

Let's wrap up this chapter with a simple but important exercise. For practice, try to change the to-do server, implemented earlier in this chapter, to work with the `gen_server` behaviour. This should be a straightforward task, but if you get stuck, the solution is in the todo_server.ex file. Be sure to either finish this exercise or analyze and understand the solution, because in future chapters you'll gradually expand on this simple server process and build a highly concurrent distributed system.

## 6.3. Summary

In this chapter, you were introduced to generic server processes. This is an important topic, and we'll continue to explore it. But first, here are some points worth repeating:

- A generic server process is an abstraction that implements tasks common to any kind of server process, such as recursion-powered looping and message passing.
- A generic server process can be implemented as a behaviour. A behaviour drives the process, whereas specific implementations can plug into the behaviour via callback modules.
- The behaviour invokes callback functions when the specific implementation needs to make a decision.
- `gen_server` is an OTP behaviour that implements a generic server process.
- A callback module for `gen_server` must implement various functions. The most frequently used ones are `init/1`, `handle_cast/2`, `handle_call/3`, and `handle_info/2`.
- You can interact with a `gen_server` process with the `GenServer` module.
- Two types of requests can be issued to a server process: calls and casts.
- A cast is a fire-and-forget type of request—a caller sends a message and immediately moves on to do something else.
- A call is a synchronous send-and-respond request—a caller sends a message and waits until the response arrives, the timeout occurs, or the server crashes.

We're finished with the basics of generic server processes. But because this topic is so important, we'll explore it in more depth in the next

chapter, where you'll learn how to implement a more involved concurrent system using `gen_server`.

# Chapter 7. Building a concurrent system

*This chapter covers*

- Working with the `mix` project
- Managing multiple to-do lists
- Persisting data
- Reasoning with processes

The concurrent examples you've seen so far relied on a single server-process instance. But typical Elixir/Erlang systems are powered by a multitude of processes, many of which are stateful server processes. It's not uncommon for a moderately complex system to run a few thousands processes, whereas larger systems may be powered by hundreds of thousands or even millions of processes. Remember that processes are cheap, so you can create them in abundance. And owing to message-passing concurrency, it's still fairly easy to reason about highly concurrent systems. Therefore, it's useful to run different tasks in separate processes. Such a highly concurrent approach can often improve scalability and reliability of your systems.

In this chapter, you'll see an example of a more involved system powered by many processes that cooperate to provide the full service. The ultimate goal is to build a distributed HTTP server that can handle many end users who are simultaneously manipulating many to-do lists. You'll do this throughout the remaining chapters and reach the final goal in chapter 12. In this chapter, you'll develop an infrastructure for handling multiple to-do lists and persisting them to the disk. But first, let's see how to manage more complex projects with the `mix` tool.

## 7.1. Working with the mix project

As the code gets more involved, it becomes increasingly clumsy to place all the modules in a single file. So, this is the right time to start working with multifile projects.

Chapter 2 briefly mentioned that Elixir ships with the `mix` tool, which you can use to create, build, and run projects as well as manage their dependencies, run tests, and create custom project-based tasks. At this point, you'll learn just enough about `mix` to know how to create and run a project. Additional `mix` features will be introduced as the need arises.

Let's use `mix` to create a project for the to-do list. Type the following at the command line:

```
$ mix new todo
```

This creates the todo folder and a project structure under it. The result is a folder that contains only a handful of files, including a readme, unit test support files, and the .gitignore file. `mix` projects are extremely simple and don't introduce a plethora of auto-generated files.

**Tip**

This book doesn't provide a detailed treatment of the `mix` tool. Instead, essential features are introduced when needed. To find out more about `mix`, check the online *Introduction to Mix* guide (http://mng.bz/Ia2l). In addition, from the command line, you can run `mix help` to get a list of available commands and `mix help command` to get detailed help for a particular command.

Once the project is in place, you can go to its folder and run `mix` tasks from there. For example, you can compile the project with `mix compile` command, or you can run tests with `mix test`.

You can also use a special form for starting `iex`, which is useful when you want to play with `mix` projects in the Elixir shell. When you run `iex -S mix`, two things happen. First, the project is compiled (just as with `mix compile`). If this is successful, the shell is started, and all modules from the project are available. The word *available* here means all generated BEAM files (binaries that represent compiled modules) are in load paths.

Using `mix`, it's possible to organize the code into multiple files and

folders. You can place .ex files under the lib folder, and they will automatically be included in the next build. You can also use arbitrarily nested subfolders under the lib folder. Of course, one .ex file may contain many modules.

There are no hard rules regarding how files should be named and organized, but there are some preferred conventions:

- You should place your modules under a common top-level alias. For example, modules might be called `Todo.List`, `Todo.Server`, and so on. This reduces the chance of module names conflicting when you combine multiple projects into a single system.
- In general, one file should contain one module. Occasionally, if a helper module is small and used only internally, it can be placed in the same file as the module using it. If you want to implement protocols for the module, you can do this in the same file as well.
- A filename should be an underscore (aka *snake*) case of the main module name it implements. For example, a `TodoServer` module resides in a todo_server.ex file in the lib folder.
- The folder structure should correspond to multipart module names. A module called `Todo.Server` should reside in the file lib/todo/server.ex.

These aren't strict rules, but they're the ones used by the Elixir project as well as many third-party libraries.

With this out of the way, let's start adding code to the project. You've already developed two modules: `TodoList` and `TodoServer`. The final version of both modules resides in the file todo_server.ex from chapter 6. Per conventions, you'll rename the modules `Todo.List` and `Todo.Server` and move them to the `todo` project. Here's what you need to do:

1. Remove the file todo/lib/todo.ex.

2. Place the `TodoList` code in the todo/lib/todo/list.ex file. Rename the module to `Todo.List`.

3. Place the `TodoServer` code in the todo/lib/todo/server.ex file. Rename the module to `Todo.Server`.

4. Replace all references to `TodoServer` with `Todo.Server` and all references to `TodoList` with `Todo.List`.

Now you can start the system with `iex -S mix` and verify that it works:

```
$ iex -S mix

iex(1)> {:ok, todo_server} = Todo.Server.start
iex(2)> Todo.Server.add_entry(todo_server,
          %{date: {2013, 12, 19}, title: "Dentist"})

iex(3)> Todo.Server.entries(todo_server, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

At this point, the to-do code is in the `mix` project, and you can continue to extend it with additional features.

#### Unit tests

Dealing with unit tests falls outside the scope of this book, but the accompanying code contains basic tests. Feel free to check them out in the test subfolder; just be aware that these example projects aren't test-driven or particularly well tested. In this book, the focus is on extremely simple code that illustrates a point. Such code is often not very testable, and improvisations (such as a lot of mocking) have been used to ensure basic correctness. To learn more about unit testing in Elixir, check out the official `ExUnit` reference (http://mng.bz/OALf).

## 7.2. Managing multiple to-do lists

This section introduces support for managing multiple to-do lists. Before starting, let's recap what you've built so far:

- A pure functional `Todo.List` abstraction

- A to-do server process that can be used to manage one to-do list for a long time

There are two approaches to extend this code to work with multiple lists:

1. Implement a `TodoListCollection` pure functional abstraction to work with multiple to-do lists. Modify `Todo.Server` to use the new abstraction as its internal state.

2. Run one instance of the existing to-do server for each to-do list.

The problem with the first approach is that you'll end up having only one process to serve all users. This approach therefore isn't very scalable. If the system is used by many different users, they will frequently block each other, competing for the same resource—a single server process that performs all tasks.

The alternative is to use as many processes as there are to-do lists. With this approach, each list is managed concurrently, and the system should be more responsive and scalable.

To run multiple to-do server processes, you need another entity—something you'll use to create `Todo.Server` instances or fetch the existing ones. That "something" must manage a state—essentially a key-value structure that maps to-do list names to to-do server pids. This state will of course be mutable (the number of lists changes over the course of time) and must be available during the server's lifetime.

Therefore, you'll introduce another process: a to-do cache. You'll run only one instance of this process, and it will be used to create and return a pid of a to-do server process that corresponds to the given name. The module will export only two functions: `start/0`, which starts the process; and `server_process/2`, which retrieves a to-do server process (its pid) for a given name, optionally starting the process if it isn't already running.

### 7.2.1. Implementing a cache

Let's begin implementing the cache process. First, copy the entire todo folder to the todo_cache folder. Then add the new file todo_cache/lib/todo/cache.ex where the code for `Todo.Cache` will reside.

Now you need to decide what the process state will be. Remember, the process will be used as a provider of to-do server pids. You give it a name, and it gives you the corresponding process. In this case, it seems reasonable to use `HashDict`, which associates to-do list names with to-do server pids. This is implemented in the following listing.

**Listing 7.1. Cache initialization (todo_cache/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  use GenServer

  def init(_) do
    {:ok, HashDict.new}
  end

  ...
end
```

With this in place, you can begin introducing the `server_process` request. You need to decide on the request's type. Because this request must return a result to the caller (a to-do server pid), there are no options—it needs to be a call. The implementation is fairly straightforward, as shown next.

**Listing 7.2. Handling the `server_process` request (todo_cache/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  ...

  def handle_call({:server_process, todo_list_name}, _, todo_servers) do
    case HashDict.fetch(todo_servers, todo_list_name) do
      {:ok, todo_server} ->                          <-- Server exists in the map
        {:reply, todo_server, todo_servers}

      :error ->                                      <-- Server doesn't exist
        {:ok, new_server} = Todo.Server.start        <-- Starts the new server
        {
          :reply,
          new_server,
          HashDict.put(todo_servers, todo_list_name, new_server)
        }
    end
  end

  ...
end
```

In this example, you use `HashDict.fetch/2` to query the map. If there is something for the given key, you return the value to the caller, leaving the state unchanged. Otherwise, you must start a server, return its pid, and insert an appropriate name-value pair in the process state.

Finally, you shouldn't forget to include interface functions, provided in the next listing.

**Listing 7.3. Interface functions (todo_cache/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  ...

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def server_process(cache_pid, todo_list_name) do
    GenServer.call(cache_pid, {:server_process, todo_list_name})
  end

  ...
end
```

The only new thing worth mentioning here is the use of `__MODULE__`. During compilation, this construct is replaced with the name of the current module. `__MODULE__` is used here as a convenience; you could write `Todo.Cache` instead, but this approach removes this minor duplication and guards the code against a possible change of the module name.

At this point the to-do cache is complete, and you can try it. Start the shell with `iex -S mix`, and do the following:

```
iex(1)> {:ok, cache} = Todo.Cache.start

iex(2)> Todo.Cache.server_process(cache, "Bob's list")
#PID<0.69.0>                                          The first retrieval creates a new process.

iex(3)> Todo.Cache.server_process(cache, "Bob's list")
#PID<0.69.0>                                          The second retrieval returns the same process.

iex(4)> Todo.Cache.server_process(cache, "Alice's list")
#PID<0.72.0>                                          A different name returns a different process.
```

The returned pid represent a to-do server process that manages a single to-do list. You can use it in the familiar way to manipulate the

list:

```
iex(5)> bobs_list = Todo.Cache.server_process(cache, "Bob's list")

iex(6)> Todo.Server.add_entry(bobs_list,
          %{date: {2013, 12, 19}, title: "Dentist"})

iex(7)> Todo.Server.entries(bobs_list, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

Of course, Alice's list isn't affected by these manipulations:

```
iex(8)> Todo.Cache.server_process(cache, "Alice's list") |>
          Todo.Server.entries({2013, 12, 19})
[]
```

Having the cache in place makes it possible for you to manage many to-do lists independently. The following session creates 100,000 to-do list servers and verifies that you have that many processes running:

```
iex(1)> {:ok, cache} = Todo.Cache.start

iex(2)> length(:erlang.processes)
36

iex(3)> 1..100_000 |>
          Enum.each(fn(index) ->
            Todo.Cache.server_process(cache, "to-do list #{index}")
          end)

iex(4)> length(:erlang.processes)
100036
```

Here you use the `:erlang.processes/0` function, which returns a list of the pids of all currently running processes.

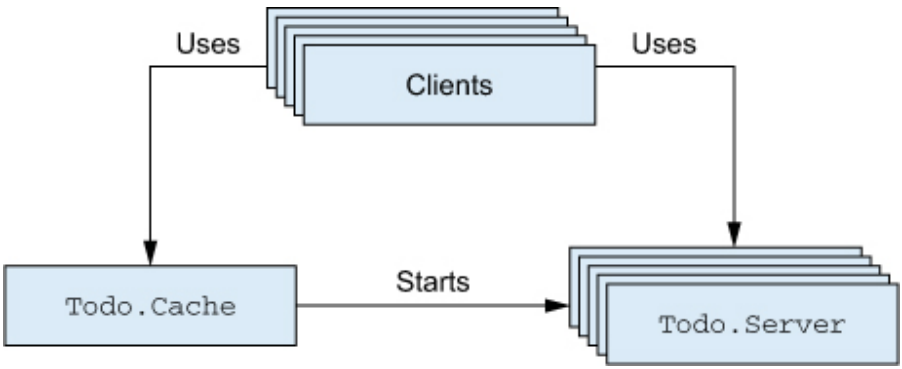
You might be puzzled as to why you initially have 36 processes running, even though you started just one. The remaining processes are those started and used internally by Elixir and Erlang.

### 7.2.2. Analyzing process dependencies

Let's reflect a bit on the current system. You've developed support for managing many to-do list instances. The end goal is to use this

infrastructure in an HTTP server. In the Elixir/Erlang world, HTTP servers typically use separate process for each request. Thus, if you have many simultaneous end users, you can expect many BEAM processes accessing your to-do cache and to-do servers. The dependency between processes is illustrated in figure 7.1.

**Figure 7.1. Cooperation of processes**




Here, each box represents a single process. The Clients boxes are arbitrary clients, such as HTTP request-handler processes. Looking at this diagram, you can immediately spot some characteristics of your system's concurrent behavior:

- Multiple (possibly a large number of) clients issue requests to the single to-do cache process.
- Multiple clients communicate with multiple to-do server processes.

The first property is the possible source of a bottleneck. Because you have only one to-do cache process, you can handle only one `server_process` request simultaneously, regardless of how many CPU resources you have.

Notice that this problem may not necessarily be significant in practice. If your `get_or_request` takes, for example, 1 microsecond, then the to-do cache can handle a load of up to 1,000,000 requests per second, which should be sufficient for most needs. But if request handling takes 100 milliseconds, then you can process only 10 requests per second, and your system won't be able to handle higher loads.

It's easy to reason about an individual process. Regardless of how many concurrent requests are coming in, a single process can handle only one request at a time. Thus, a process is good enough if its request-handling rate is at least equal to the incoming rate. Otherwise, you have to either optimize the process or do other interventions.

For this specific case, the to-do cache performs very simple operatio a `HashDict` lookup followed by an optional process creation and `HashDict` update. According to a quick test on my machine, for 1 million to-do lists, it takes about 25 microseconds to start a new to-do server and put it in the hash, or 5 microseconds to fetch the existing one. This should be sufficient for a load of at least 40,000 requests/sec, which seems like reasonable performance for the initial attempt.

There is an added benefit to the sequential nature of processes. Because a process runs only one request at a time, its internal state is consistent. You know there can't be multiple simultaneous updates of the process state, which makes race conditions in a single process impossible.

**Tip**

If you need to make sure part of the code is synchronized—that is, that there are no multiple simultaneous executions of critical code—it's best to run that code in a dedicated process. When multiple clients want this code to run, they issue a request to that process. The process then serves as a synchronization point, making sure the critical code is run in a single process.

I need to say a few words about client interactions with to-do servers. Once a client gets a to-do server pid, the list manipulation runs concurrently to all other activities in the system. Because you can expect that list manipulations will be fairly involved, it's beneficial to run those operations concurrently. This is the part where your system is concurrent and scalable—it can manipulate multiple lists, using as many resources as possible.

Also recall from chapter 5 that a process waiting for a message is suspended and doesn't waste CPU resources. Thus, regardless of the number of processes, only those that are actually doing computations

consume CPU. In this case, this means a client process doesn't use CPU while it waits for a to-do server to finish.

Finally, you can be sure that a single list can't be modified by two simultaneous clients. Recall that the list is managed by a single process. Even if a million clients try to modify the same list, their requests will be serialized in the corresponding to-do server and handled one by one.

Now you have a basic system that you can use to manipulate many to-do lists. It's time to include basic persistence so your data can outlive server restarts.

## 7.3. Persisting data

In this section, you'll extend to-do cache and introduce basic data persistence. The focus here isn't so much on the persistence itself, but rather on exploring the process model—how you can organize your system into various server processes, analyze dependencies, and identify and address bottlenecks. You'll start with the code from the `todo_cache` project and extend it gradually. For data persistence, you'll use simple disk-based persistence, encoding the data into the Erlang external term format. The complete solution is in the persistable_todo_cache folder.

### 7.3.1. Encoding and persisting

To encode an arbitrary Elixir/Erlang term, you use the `:erlang.term_to_binary/1` function, which accepts an Erlang term and returns an encoded byte sequence as a binary value. The input term can be of arbitrary complexity, including deep hierarchies of nested lists and tuples. The result can be stored to disk, retrieved at a later point, and decoded to an Erlang term with the inverse function `:erlang.binary_to _term/1`.

Equipped with this knowledge, you'll introduce another process: a

database powered by the `Todo.Database` module. This will be a simple process that supports two requests: `store` and `get`. While storing data, clients will provide a key and the corresponding data. The data will be stored in the file that bears the same name as the key. This approach is far from perfect and is error prone, but it's simple enough to let us focus on the concurrency aspect of the problem. The full implementation of the database process is given in the following listing.

**Listing 7.4. Database process (persistable_todo_cache/lib/todo/database.ex)**

```
defmodule Todo.Database do
  use GenServer

  def start(db_folder) do
    GenServer.start(__MODULE__, db_folder,
     name: :database_server                    <-- Locally registers the process
     )
  end

  def store(key, data) do
    GenServer.cast(:database_server, {:store, key, data})
  end

  def get(key) do
    GenServer.call(:database_server, {:get, key})
  end

  def init(db_folder) do
    File.mkdir_p(db_folder)                    <-- Makes sure the folder exists
    {:ok, db_folder}
  end


  def handle_cast({:store, key, data}, db_folder) do
     file_name(db_folder, key)
     |> File.write!(:erlang.term_to_binary(data))

     {:noreply, db_folder}
   end

  def handle_call({:get, key}, _, db_folder) do
     data = case File.read(file_name(db_folder, key)) do
       {:ok, contents} -> :erlang.binary_to_term(contents)
       _ -> nil
     end

     {:reply, data, db_folder}
   end

  defp file_name(db_folder, key), do: "#{db_folder}/#{key}"
end
```

Stores the data

Reads the data

This is mostly a synthesis of techniques mentioned earlier. The database process receives a database folder via the parameter. The folder location is then kept as the process state. Notice how the

process is locally registered under an alias; this keeps things simple and relieves you from passing around the `Todo.Database` pid. Of course, a downside is that you can run only one instance of the database process.

It's worth noting that the `store` request is a cast, whereas `get` is a call. In this implementation, I decided to turn `store` into a cast, because the client is not interested in a response. Using casts promotes scalability of the system because the caller issues a request and goes about its business.

But a huge downside of a cast is that the caller can't know whether the request was successfully handled. In fact, the caller can't even be sure that the request reached the target process. This is a property of casts. Casts promote overall availability by allowing client processes to move on immediately after a request is issued. But this comes at the cost of consistency, because you can't be confident about whether a request has succeeded. In this example, let's start with the `store` request being a cast. This makes the entire system more scalable and responsive, with the downside being that you can't make true guarantees that all changes have been persisted.

During initialization, you use `File.mkdir_p/1` to create the given folder if it doesn't exist. The data is stored by encoding the given term to the binary and then persisting it to the disk. Data fetching is an inverse of storing. If the given file doesn't exist on the disk, you return `nil`. Notice that you use the `File.write!/2` and `File.read/1` functions for file operations.

### 7.3.2. Using the database

With the database process in place, it's time to use it from your existing system. You have to do three things:

1. Ensure that a database process is started.
2. Persist the list on every modification.

3. Try to fetch the list from a disk during the first retrieval.

To start the server, you'll plug into the `Todo.Cache.init/1` function. This is a quick hack, but it's sufficient for the moment. The modification is provided next.

**Listing 7.5. Starting the database (persistable_todo_cache/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  ...

  def init(_) do
    Todo.Database.start("./persist/")
    {:ok, HashDict.new}
  end

  ...
end
```

Here you use the persist subfolder of the current folder as the place where you store data.

#### Storing the data

Next you have to persist the list after it's modified. Obviously, this must be done from the to-do server. But remember that the database's `store` request requires a key. For this purpose, you'll use the to-do list name. As you may recall, this name is currently maintained only in the to-do cache, so you must propagate it to the to-do server as well. This means extending the to-do server state to be in the format `{list_name, todo_list}`. The code isn't provided here, but these are the corresponding changes:

- `Todo.Server.start` now accepts the to-do list name and passes it to `GenServer .start/2`.
- `Todo.Server.init/1` uses this parameter and keeps the list name in the process state.
- `Todo.Server.handle_*` callbacks are updated to work with the new state format.
- While starting the new to-do server, the cache process passes the list name.

After these modifications, the to-do server knows its own name. Now it's trivial to persist the data, as shown in the following listing.

**Listing 7.6. Persisting the data (persistable_todo_cache/lib/todo/server.ex)**

```
defmodule Todo.Server do
  ...
  def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
    new_state = Todo.List.add_entry(todo_list, new_entry)
    Todo.Database.store(name, new_state)                    <-- Persists the data
    {:noreply, {name, new_state}}
  end

  ...
end
```

You can immediately test whether this works. Run `iex -S mix`, and try the following:

```
iex(1)> {:ok, cache} = Todo.Cache.start
iex(2)> bobs_list = Todo.Cache.server_process(cache, "bobs_list")
iex(3)> Todo.Server.add_entry(bobs_list,
          %{date: {2013, 12, 19}, title: "Dentist"})
```

If all goes well, there should be a file named persist/bobs_list on the disk.

#### Reading the data

All that's left to do is to read the data from the disk when the server is started. You'll use a simplistic implementation here, as illustrated next.

**Listing 7.7. Reading data (persistable_todo_cache/lib/todo/server.ex)**

```
defmodule Todo.Server do
  ...

  def init(name) do
    {:ok, {name, Todo.Database.get(name) || Todo.List.new}}
  end

  ...
end
```

Here you try to fetch the data from the database, and you resort to the empty list if there's nothing on the disk. This is a simplistic approach that works for this case, but you should generally be careful about possibly long-running `init/1` callbacks. Recall that `GenServer.start` returns only after the process has been initialized. Consequently, a long-running `init/1` function will cause the creator process to block. In this case, a long initialization of a to-do server will

block the cache process, which is used by many clients.

To circumvent this problem, there's a simple trick. You can use `init/1` to send yourself an internal message and then initialize the process state in the corresponding `handle_info` callback:

```
def init(params) do
  send(self, :real_init)          <-- Sends itself a message
  {:ok, nil}                      <-- You're not really initialized here.
end

...

def handle_info(:real_init, state) do
  ...                             <-- Performs a long initialization
end
```

By sending yourself a message, you place a request in your own message queue. Then you return immediately, which makes the corresponding `GenServer.start` return, and the creator process can continue running other code. In the meantime, the process loop starts and immediately handles the first message, which is `:real_init`.

This technique will generally work as long as your process isn't registered under a local alias. If the process isn't registered, someone has to know its pid to send it a message, and that pid will only be known after `init/1` has finished. Hence, you can be sure that the message you send to yourself is the first one being handled.

But if the process is registered, there is a chance that someone else will put the message in their queue first by referencing the process via symbolic alias. This can happen because at the moment `init/1` is invoked, the process is already registered under the alias (due to the inner workings of `:gen_server` module). There are a couple of workarounds for this problem, the simplest one being to not use the `name: :some_alias` option and opt instead for manual registration of the process in the `init/1` callback after the message to self is sent:

```
def init(params) do
  ...
  send(self, :real_init)
  register(self, :some_alias)     <-- Manual alias registration
end
```

This is a bit of a hack, so use this technique judiciously and only in situations when you don't want to block the caller due to long initialization.

In any case, the to-do server now reads data from the database on creation. You can immediately test this. If you have the previous shell session open, close it, and start the new one. Then try the following:

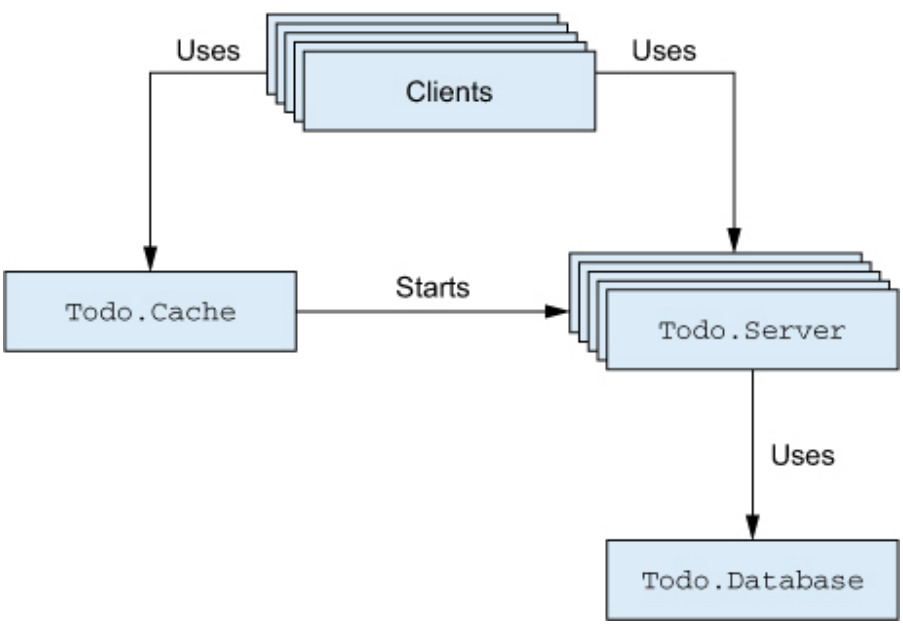
```
iex(1)> {:ok, cache} = Todo.Cache.start
iex(2)> bobs_list = Todo.Cache.server_process(cache, "bobs_list")

iex(3)> Todo.Server.entries(bobs_list, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 1, title: "Dentist"}] <-- Data retrieved from the disk
```

As you can see, your to-do list contains data, which proves that deserialization works.

### 7.3.3. Analyzing the system

Let's analyze how the new version of the system works. The process interaction is presented in figure 7.2.

**Figure 7.2. Process dependencies**



You introduced just one process, but it can have a negative impact on the entire system. Recall that the database performs term encoding/decoding and, even worse, disk I/O operations. Depending on the load and list sizes, this can affect performance badly. Let's recall all the places where database requests are issued:

```
defmodule Todo.Server do
  ...
  def init(name) do
    {:ok, {name, Todo.Database.get(name) || Todo.List.new}}
  end

  ...

  def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
    ...
    Todo.Database.store(name, todo_list)
    ...
  end
  ...
end
```

Synchronous request

Asynchronous request

The `store` request may not seem problematic from the client-side perspective, because it's an asynchronous cast. A client issues a `store` request and then goes about its business. But if requests to the database come in faster than they can be handled, the process mailbox will grow and increasingly consume memory. Ultimately, the entire system may experience significant problems, resulting in possible termination of the BEAM OS process.

The `get` request can cause additional problems. It's a synchronous call, so the to-do server waits while the database returns the response. While it's waiting for the response, this to-do server can't handle new messages. What's worse, because this is happening from initialization, the cache process is blocked until the list data is retrieved, which ultimately may render the entire system useless under a heavier load.

It's worth repeating that the synchronous call won't block indefinitely. Recall that it has a default timeout of 5 seconds, and you can configure it to be less, for better responsiveness. Still, when a request times out, it isn't removed from the receiver's mailbox. A request is a message that is placed in the receiver's mailbox. A timeout means you give up waiting on the response, but the message remains in the receiver's mailbox and will be processed at some point.

### 7.3.4. Addressing the process bottleneck

It's obvious that you should address the bottleneck introduced by the singleton database process. There are many approaches, but here we'll discuss only a couple of them.

#### Bypassing the process

The simplest possible way to eliminate the process bottleneck is to bypass the process. You should ask yourself—does this need to be a process, or can it be a plain module?

There are various reasons for running a piece of code in a dedicated server process:

- The code must manage a long-living state.
- The code handles a kind of a resource that can and should be reused: for example, a TCP connection, database connection, file handle, pipe to an OS process, and so on.
- A critical section of the code must be synchronized. Only one process may run this code in any moment.

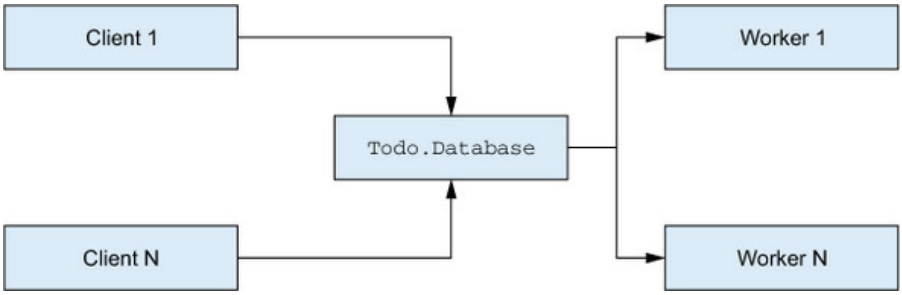
If none of these conditions are met, you probably don't need a process and can run the code in client processes, which will completely eliminate the bottleneck and promote parallelism and scalability.

Unfortunately, your database operations must be synchronized on individual items. Each item is stored in its own corresponding file, and you shouldn't allow multiple concurrent read/writes of the same item. Thus this approach won't work in this case.

#### Handling requests concurrently

Another option is to keep the database process and make it handle database operations concurrently. This is useful when requests depend on a common state but can be handled independently. The idea is illustrated in figure 7.3.

**Figure 7.3. Handling requests concurrently**




As you can see, each request still is serialized through the central server process. But this server process spawns one-off worker processes that perform the actual request handling. If you keep the code in the database process short and fast, you'll get to keep a high degree of scalability with many workers running concurrently.

To implement this, you must run each database operation in a spawned one-off process. For casts, this means transforming the body of the handler:

```
def handle_cast({:store, key, data}, db_folder) do
  spawn(fn ->
    file_name(db_folder, key)
    |> File.write!(:erlang.term_to_binary(data))
  end)

  {:noreply, db_folder}
end
```

Handled in a spawned process

The handler function spawns the new worker process and immediately returns. While the worker is running, the database process can accept new requests.

For synchronous calls, this approach is slightly more complicated because you have to return the response from the spawned worker process:

```
def handle_call({:get, key}, caller, db_folder) do
  spawn(fn ->                                          <-- Spawns the reader
    data = case File.read(file_name(db_folder, key)) do
      {:ok, contents} -> :erlang.binary_to_term(contents)
      _ -> nil
    end

    GenServer.reply(caller, data)        <-- Responds from the spawned process
  end)

  {:noreply, db_folder}          <-- No reply from the database process
end
```

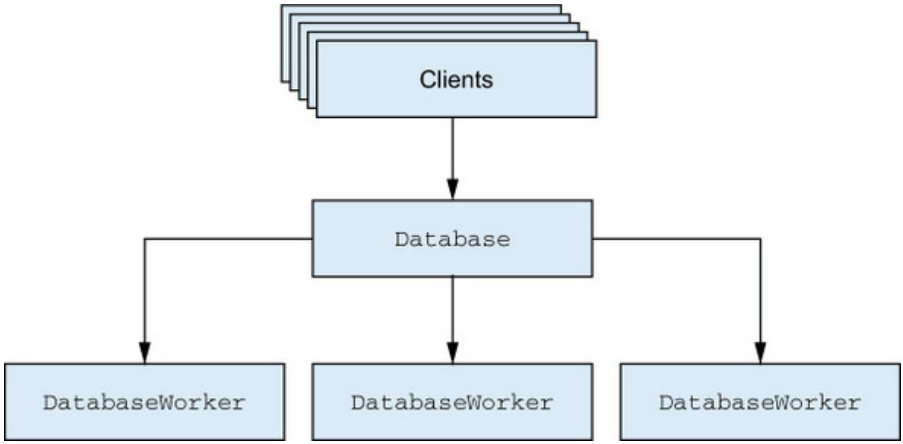
The server process spawns another, worker process, and then returns `{:noreply, state}`, indicating to `gen_server` that you won't reply at this point. In the meantime, the spawned process handles the request and reports back to the caller with `GenServer.reply/2`. This is one situation where you need to use the second argument of `handle_call/3`: the caller pid and the unique ID of the request. This information is used in the spawned process to send the response message to the caller.

This technique keeps the processing in the database process short while still allowing concurrent execution of database operations. But the problem with this approach is that concurrency is unbound. If you have 100,000 simultaneous clients, then you'll issue that many concurrent I/O operations, which may negatively affect the entire system.

#### Limiting concurrency with pooling

A typical remedy for this is to introduce pooling. For example, your database process might create three worker processes and keep their pids in its internal state. When a request arrives, it's delegated to one of the worker processes, perhaps in a round-robin fashion or with some other load-distribution strategy. The idea is presented in figure 7.4.

**Figure 7.4. Pooling database operations**




Of course, all requests still arrive at the database process first, but they're quickly forwarded to one of the workers. Essentially, this technique keeps the concurrency level under control and works best when dealing with resources that can't cope with a large number of concurrent requests.

This technique also allows you to synchronize operations. Remember, you want to avoid simultaneous database operations on a single item. This can easily be done by altering a strategy that selects the worker process. The idea is to make requests for the same item end up in the same worker. The worker then becomes the synchronization point for the single item. This can be as simple as computing a numerical hash of the key, normalizing it to fall in the range `0..N-1`, and forwarding the request to the appropriate worker. Notice that due to limited pool size, each worker handles multiple items, but a single item is always handled in the same worker.

This approach will work correctly in this example, so it's the one you'll use. Of course, in a different situation, some other approach might work best. The point of this analysis is to illustrate how to think in terms of processes. Always keep in mind that multiple processes run concurrently, whereas a single process handles requests sequentially. If computations can safely run in parallel, you should consider running them in separate processes. In contrast, if an operation must be synchronized, you'll want to run it in a single process.

**Database connection pool**

Increasing the number of concurrent disk-based operations doesn't in reality yield significant improvements and may hurt performance. In this sense, the optimizations serve more as a didactic example than an efficient solution. But in real life, you would probably talk to a database that is able to handle multiple concurrent requests efficiently. In such a case, you would typically need to constrain the number of simultaneous database operations. And this is the purpose of a pool of processes.

Of course, there's no need to implement such a pool yourself. A couple of generic pool libraries are available for the Elixir/Erlang ecosystem, one of the more popular being poolboy (https://github.com/devinus/poolboy). Depending on which database library you're using, either you'll need to combine it with poolboy (or another pooling solution), or this will be done by the library (as is the case, for example, with the Ecto library [https://github.com/elixir-lang/ecto], which internally relies on poolboy).

### 7.3.5. Exercise: pooling and synchronizing

Now it's time for you to practice a bit. This exercise introduces pooling and makes the database internally delegate to three workers that perform the actual database operations. Moreover, there should be per-key (to-do list name) synchronization on the database level. Data with the same key should always be treated by the same worker.

Here are some pointers for doing this:

**1**. Start with the existing solution, and migrate it gradually. Of the existing code, the only thing that needs to change is the `Todo.Database` implementation. This means you don't have to touch any of other modules.

**2**. Introduce a `Todo.DatabaseWorker` module. It will be almost a copy-paste of the current `Todo.Database`. But the process must not be registered under an alias, because you need to run multiple instances.

3. `Todo.Database` will receive a significant rewrite, but its interface must remain the same. This means it still implements a locally registered process that is used via these functio `start/1`, `get/1`, and `store/2`.

4. During `Todo.Database` initialization, start three workers, and store their pids in a `HashDict`, using zero-based indexes as values.

5. In `Todo.Database`, implement a single request, `get_worker`, that will return a worker's pid for a given key.

6. `get_worker` should always return the same worker for the same key. The easiest way to do this is to compute the key's numerical hash and normalize it to fall in the range [0, 2]. This can be done by calling `:erlang.phash2(key, 3)`.

7. The interface functions `get` and `store` of `Todo.Database` internally call `get_worker` to obtain the worker's pid and then forward to interface functions of `DatabaseWorker` using the obtained pid as the first argument.

Always try to work in small steps, and test as often as possible. For example, once you implement `Todo.DatabaseWorker`, you can immediately start `iex -S mix` and try it in isolation.

The same goes for `Todo.Database`. First you can initialize the state without implementing a request handler. Call `IO.inspect` from `init/1` to verify that the state is correct. Then implement `get_worker`, and test that it works in the shell. Finally, add interface functions and test the entire system.

How can you be sure that requests for different keys are running in different processes? You can again use `IO.inspect` and, from within the worker, print the pid and the key using something like `IO.inspect "#{self}: storing #{key}"`. Use `IO.inspect` extensively. It's your friend and can help you significantly during development.

If you get stuck, the complete solution is in the todo_cache_pooling folder. Make sure you understand the solution, because you'll continue

extending this version in subsequent chapters.

## 7.4. Reasoning with processes

You have seen various examples of server processes in action. The point of these examples has been to demonstrate how simple it is to reason about an involved concurrent system.

A server process is a simple entity, something like a concurrent object. From within, it's a sequential thing that accepts and handles requests, optionally managing an internal state. From the outside, it's a concurrent agent that exposes a well-defined communication interface.

Another way to look at server processes is to think of them as services. Each process is like a small service that is responsible for a single task. In the to-do example, there is a to-do server that handles a distinct to-do list. Different lists are handled by different to-do servers, which makes the system more efficient. But a single list is always handled by the same process, which eliminates race conditions and keeps consistency. The to-do cache is a service that provides to-do servers based on the to-do list names. Finally, the database process deals with database related requests. Internally, it distributes the work over a limited pool of worker processes, making sure the same item is always handled by the same worker.

Those services (processes) are mostly independent, but in some cases they need to cooperate. For this purpose, you can use calls and casts. Obviously, when a client needs a response, you should use calls. But even when a response isn't needed, calls can sometimes be a better fit. The main problem with a cast is that it's a fire-and-forget kind of request, so the caller doesn't get any guarantees. You can't be sure that the request has reached the target, and you most certainly don't know about its outcome.

Essentially, both types have benefits and downsides. Casts promote system responsiveness (because a caller isn't blocked) at the cost of reduced consistency (because a caller doesn't know about the outcome

of the request). On the other hand, calls promote consistency (a caller gets a response) but reduce system responsiveness (a caller is blocked while waiting for a response).

Finally, calls can also be used to apply back pressure to client processes. Because a call blocks a client, it prevents the client from generating too much work. The client becomes synchronized with the server and can never produce more work than the server can handle. In contrast, if you use casts, clients may overload the server, and requests may pile up in the message box and consume memory. Ultimately, you may run out of memory, and the entire VM may be terminated.

Which approach is a better fit depends on the specific situation and circumstances. If you're unsure, then it's probably better to start with a call, because it's more consistent. You can then consider switching to casts in places where you establish that calls hurt performance and system responsiveness.

## 7.5. Summary

We're finished with our detailed exploration of concurrent systems. In this chapter, you've implemented a basic back end for managing multiple to-do lists. You'll continue to expand this system throughout the book, until you implement a distributed HTTP server. For now, let's wrap up by summarizing this chapter's most important points:

- When a system needs to perform various tasks, it's often beneficial to run different tasks in separate processes. Doing so promotes scalability and fault-tolerance of the system.
- A process is internally sequential and handles requests one by one. A single process can thus keep its state consistent, but it can also cause a performance bottleneck if it serves many clients.
- Carefully consider calls versus casts. Calls are synchronous and therefore block the caller. If the response isn't needed, casts may improve performance at the expense of reduced guarantees, because a client process doesn't know the outcome.
- You can use `mix` projects to manage more involved systems

that consist of multiple modules.

This concludes our practical tour of concurrent systems. Next, we'll look at how processes make it possible to build reliable, fault-tolerant systems.

# Chapter 8. Fault-tolerance basics

*This chapter covers*

- Runtime errors
- Errors in concurrent systems
- Supervisors

Fault tolerance is a first-class concept in BEAM. The ability to develop reliable systems that can operate even when faced with runtime errors is what brought us Erlang in the first place.

The aim of fault tolerance is to acknowledge the existence of failures, minimize their impact, and ultimately recover without human intervention. In a sufficiently complex system, many things can go wrong. Occasional bugs will happen, components you're depending on may fail, and you may experience hardware failures. A system may also become overloaded and fail to cope with an increased incoming request rate. Finally, if a system is distributed, you can experience additional issues such as a remote machine becoming unavailable, perhaps due to a crash, or a broken network link.

It's hard to predict everything that can go wrong, so it's better to face the harsh reality that anything can fail. Regardless of which part of the system happens to fail, you shouldn't take down the entire system; you want to be able to provide at least some service. For example, if the database server becomes unreachable, you can still serve data from the cache. You might even queue incoming store requests and try to resolve them later, when the connection to the database is reestablished.

Another thing you need to do is detect failures and try to recover from them. In the previous example, the system may try to reconnect to the database until it succeeds, and then resume providing full service.

These are the properties of a resilient, self-healing system. Whatever goes wrong (and remember, anything can go wrong), the system

should keep providing as much service as possible and fully recover as soon as possible.

Such thinking significantly changes the approach to error handling. Instead of obsessively trying to reduce the number of errors, your priority should be to minimize their effects and recover from them automatically. In a system that has to run continuously, it's better to experience many isolated errors than to encounter a single error that takes down the entire system.

It's somewhat surprising that the core tool for error handling is concurrency. In the BEAM world, two concurrent processes are completely separated: they share no memory, and a crash in one process can't by default compromise the execution flow of another. Process isolation allows you to confine negative effects of an error to a single process or a small group of related processes, which keeps most of the system functioning normally.

Of course, when a process crashes, you usually want to detect this state and do something about it. In this chapter, you'll learn the basic techniques of detecting and handling errors in a concurrent system. Then, in chapter 9, we'll expand on this knowledge and implement fine-grained error isolation. Let's start with a bit of theory about runtime errors.

## 8.1. Runtime errors

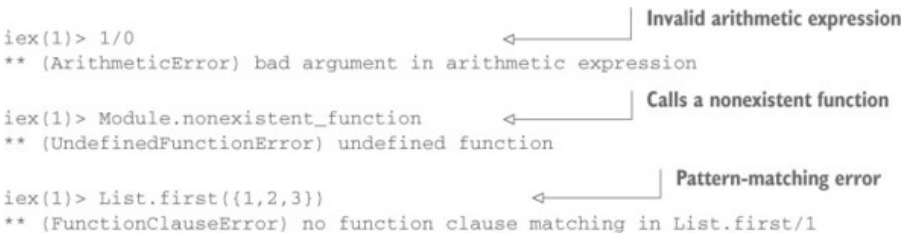
In previous chapters, we loosely mentioned that in various situations, an error is raised. One of the most common examples is a failed pattern match. If a match fails, an error is raised. Another example is a synchronous `GenServer.call`. If the response message doesn't arrive in a given time interval (five seconds by default), a runtime error happens. There are many other examples, such as invalid arithmetic operations (such as division by zero), invocation of a nonexistent function, and explicit error signaling.

When a runtime error happens, execution control is transferred up the call stack to the error-handling code. If you didn't specify such code,

then the process where the error happened is terminated. All other processes by default run unaffected. Of course, there are means to intercept and handle errors, and these resemble the familiar `try-catch` constructs you probably know from other languages.

### 8.1.1. Error types

BEAM distinguishes three types of runtime errors: *errors*, *exits*, and *throws*. Here are some typical examples of errors:




You can also *raise* your own error by using the `raise/1` macro, passing an error string:

```
iex(1)> raise("Something went wrong")
** (RuntimeError) Something went wrong
```

If your function explicitly raises an error, you should append the `!` character to its name. This is a convention used in Elixir standard libraries. For example, `File.open!` raises an error if a file can't be opened:

```
iex(1)> File.open!("nonexistent_file")
** (File.Error) could not open non_existing_file:
  no such file or directory
```

In contrast, `File.open` (notice the lack of `!`) just returns the information that the file couldn't be opened:

```
iex(1)> File.open("nonexistent_file")
{:error, :enoent}
```

Notice that in this snippet there's no runtime error. `File.open` returns a result, which the caller can handle in some way.

Another type of error is the *exit*, which is used to deliberately terminate a process. To exit the current process, you can call `exit/1`, providing an *exit reason*:

```
iex(2)> spawn(fn ->
          exit("I'm done")          <-- Exits the current process
          IO.puts "This doesn't happen"
        end)
```

The exit reason is an arbitrary term that describes why you're terminating the process. As you'll see later, it's possible for some other process to detect a process crash and obtain this exit reason.

The final runtime error type is a *throw*. To issue a throw, you can call `throw/1`:

```
iex(3)> throw(:thrown_value)
** (throw) :thrown_value
```

The purpose of throws is to allow non-local returns. As you saw in chapters 3 and 4, Elixir programs are organized in many nested function calls. In particular, loops are implemented as recursions. The consequence is that there are no constructs such as `break`, `continue`, and `return`, which you've probably seen in other languages. When you're deep in a loop, it's not trivial to stop the loop and return a value. Throws can help with this. You can throw a value and catch it up the call stack. But using throws for control flow is hacky, somewhat reminiscent of `goto`, and you should avoid this technique as much as possible.

### 8.1.2. Handling errors

It is of course possible to intercept any kind of error (error, exit, or throw) and do something about it. The main tool for this is the `try` construct. Here's how to run some code and catch errors:

```
try do
  ...
catch error_type, error_value ->
  ...
end
```

This works roughly the same as what you've probably seen in other languages. The code in the `do` block is executed, and, if an error happens, execution is transferred to the `catch` block.

Notice that two things are specified in the `catch`. An `error_type` will contain an atom `:error`, `:exit`, or `:throw`, indicating the type of an error that has occurred. And an `error_value` will contain error-specific information such as a value that was thrown or an error that was raised.

Let's play with this a bit by writing a helper lambda to make it easier to experiment with errors:

```
iex(1)> try_helper = fn(fun) ->
          try do
            fun.()
            IO.puts "No error."
          catch type, value ->
            IO.puts "Error\n  #{inspect type}\n  #{inspect value}"
          end
        end
```

This helper lambda takes a function as its argument, calls this function in a `try`, and reports the type of an error and the corresponding value. Let's try it out:

```
iex(2)> try_helper.(fn -> raise("Something went wrong") end)
Error
  :error                                                    <-- Error type
  %RuntimeError{message: "Something went wrong"}            <-- Error value
```

Notice how the string message is wrapped in a `RuntimeError` struct. This is an Elixir-specific decoration done from within the `raise/1` macro. If you want to raise a naked, undecorated error, you can use Erlang's `:erlang.error/1` and provide an arbitrary term. The resulting error value will be the term you've raised.

If you attempt to throw a value, you'll get a different error type:

```
iex(3)> try_helper.(fn -> throw("Thrown value") end)
Error
  :throw
  "Thrown value"
```

And of course, calling `exit/1` has its own type:

```
iex(4)> try_helper.(fn -> exit("I'm done") end)
Error
  :exit
  "I'm done"
```

Remember that in Elixir, everything is an expression that has a return value. With `try`, a return value is the result of the last executed statement—either from the `do` block or, if an error was raised, from the `catch` block:

```
iex(5)> result =
          try do
            throw("Thrown value")
          catch type, value -> {type, value}
          end

iex(6)> result
{:throw, "Thrown value"}
```

It's also worth noting that the `type` and `value` specified in the `catch` block are patterns. If you want to handle a specific type of an error, you can do this by providing corresponding patterns.

For example, let's say you want to immediately return a value from inside a deep nested loop. You can invoke the following:

```
throw({:result, some_result})
```

Then, somewhere up the call stack, you handle this particular throw:

```
try do
  ...
catch :throw, {:result, x} -> x
end
```

In this example, you only match for a specific runtime error: a throw in the form `{:result, x}`. If anything else is raised, you won't catch it, and an error will be propagated further up the call stack. Of course, if no one handles the error, then the process terminates.

Because `catch` is a pattern match, multiple clauses can be specified, just as you've seen with `case` and `receive` constructs:

```
try do
  ...
catch
  type_pattern_1, error_value_1 ->
    ...
  type_pattern_2, error_value_2 ->
    ...

  ...
end
```

The block under the first pattern that matches a raised error is invoked, and the result of the last statement is returned.

If you want to catch anything, you can use the `type, value` pattern, or `_, _` if you're not interested in values. These patterns will handle any error that can occur.

It's also possible to specify code that should always be executed after the `try` block, regardless of whether an error was raised:

```
iex(7)> try do
          raise("Something went wrong")
        catch
          _,_ -> IO.puts "Error caught"
        after
          IO.puts "Cleanup code"        <-- Always executed
        end

Error caught
Cleanup code
```

Because it's always executed, the `after` block is useful to clean up resources—for example, to close an open file.

It's worth noting that the `after` clause doesn't affect the result of the entire `try` construct. The result of `try` is the result of the last statement either from the `do` block or from the corresponding `catch` block if something was caught.

**Try and tail calls**

You may recall the tail-call optimization from chapter 3. If the last

thing a function does is call another function (or itself), then a simple jump will occur without a stack push. This optimization isn't possible if the function call resides in a `try` construct. This is fairly obvious, because the last thing a function does is a `try` block, and it won't finish until its `do` or `catch` block is done. Consequently, whatever is called in `try` isn't the last thing a function does and is therefore not available for tail-call optimization.

There's much more to signaling and handling runtime errors. Elixir provides some abstractions on top of this basic mechanism. You can define custom errors via a `defexception` macro (see http://mng.bz/4ggy) and handle them in a slightly more elegant fashion. The `try` construct also has a couple other features that we didn't discuss. You should definitely research the official `try` documentation (http://mng.bz/mE2S) as well as the corresponding "Getting Started" section (http://elixir-lang.org/getting_started/17.html).

What we've presented here are the core concepts of runtime errors. All other extensions supported by Elixir eventually boil down to these concepts and have the same properties:

- A runtime error has a type, which can be `:error`, `:exit`, or `:throw`.
- A runtime error also has a value, which can be any arbitrary term.
- If a runtime error isn't handled, the corresponding process will terminate.

But compared to languages such as C++, C#, Java, and JavaScript, there's much less need to catch runtime errors. A more common idiom is to let the process crash and then do something about it (usually, restart the process). This approach may seem hacky, but there is a reasoning behind it. In a complex system, most bugs are flushed out in the testing phase. The remaining bugs mostly fall into a so-called *Heisenbug* category—unpredictable errors that occur irregularly in special circumstances and are hard to reproduce. The cause of such errors usually lies in corruptness of the state. Therefore, a reasonable remedy for such errors is to let the process crash and start another one.

This may help, because you're getting rid of the process state (which may be corrupt) and starting with a clean state. In many cases, doing so resolves the immediate problem. Of course, the error should be logged so you can analyze it later and detect the root cause. But in the meantime, you can recover from an unexpected failure and continue providing service. This is a property of a self-healing system.

Don't worry if this discussion seems vague. This approach to error handling, also known as *letting it crash*, will be explained in detail throughout this chapter and the next. In the following section, we'll look at the basics of error handling in concurrent systems.

## 8.2. Errors in concurrent systems

Concurrency plays a central role in building fault-tolerant, BEAM-based systems. This is due to the total isolation and independence of individual processes. A crash in one process won't affect the others (unless you explicitly want it to). Here's a quick demonstration:

```
iex(1)> spawn(fn ->                          <-- Starts process 1
          spawn(fn ->                          <-- Starts process 2
            :timer.sleep(1000)
            IO.puts "Process 2 finished"
          end)

          raise("Something went wrong")      <-- Raises an error from within process 1
        end)
```

Running this yields the following output:

```
17:36:20.546 [error] Error in process <0.61.0>      <-- Error logger output
...
Process 2 finished                           <-- Output of process 2
```

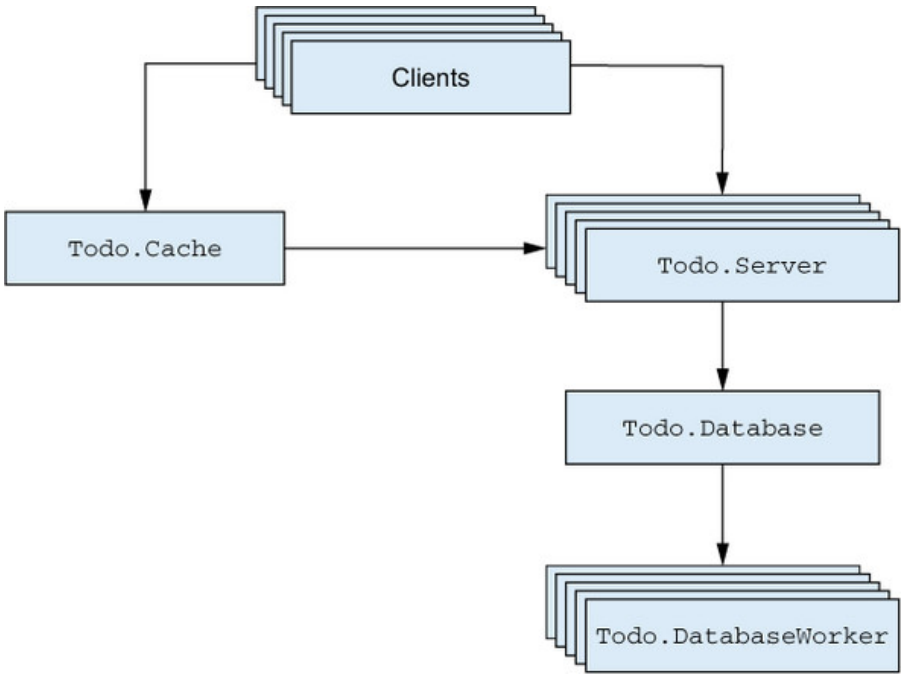
As you can see, the execution of process 2 goes on despite the fact that process 1 crashes. Of course, information about the crash of process 1 is printed to the screen, but the rest of the system—including process 2 and the `iex` shell prompt—runs normally.

Furthermore, because processes share no memory, a crash in one process won't leave memory garbage that might corrupt another

process. Therefore, by running independent actions in separate processes, you automatically ensure isolation and protection.

You already benefit from process isolation in this book's example to-do system. Recall the current architecture, shown in figure 8.1.

**Figure 8.1. Isolating errors in the to-do system**




All the boxes in the figure are BEAM processes. A crash in a single to-do server doesn't affect operations on other to-do lists. A crash in `Todo.Database` doesn't block cached reads that take place in to-do server processes.

Of course, this isolation isn't enough by itself. As you can see in figure 8.1, processes often communicate with each other. If a process isn't running, then its clients can't use its services. For example, if the database process goes down, then to-do servers can't query it. What's worse, modifications to the to-do list won't be persisted. Obviously this isn't desirable behavior, and you must have a way of detecting a process crash and somehow recovering from it.

### 8.2.1. Linking processes

A basic primitive for detecting a process crash is the concept of *links*. If two processes are linked, and one of them terminates, the other process receives an *exit signal*—a notification that a process has crashed.

An exit signal contains the pid of the crashed process and the *exit reason*—an arbitrary Elixir term that provides a description of why the process has terminated. In the case of a normal termination (the spawned function has finished), the exit reason is the atom `:normal`. By default, when a process receives an exit signal from another process, and that signal is anything other than `:normal`, the linked process terminates as well. In other words, when a process terminates abnormally, the linked process is also taken down.

One link connects exactly two processes and is always bidirectional. To create a link, you can use `Process.link/1`, which connects the current process with another process. More often, a link is created when you start a process. You can do this by using `spawn_link/1`, which spawns a process and links it to the current one. Let's verify this. In the following example, you again spawn two processes, this time linking them together. Then you take down one process:

```
iex(1)> spawn(fn ->
          spawn_link(fn ->                <-- Starts process 2 and links it to process 1
            :timer.sleep(1000)
            IO.puts "Process 2 finished"
          end)

          raise("Something went wrong")
        end)
```

Not surprisingly, this example gives the following output:

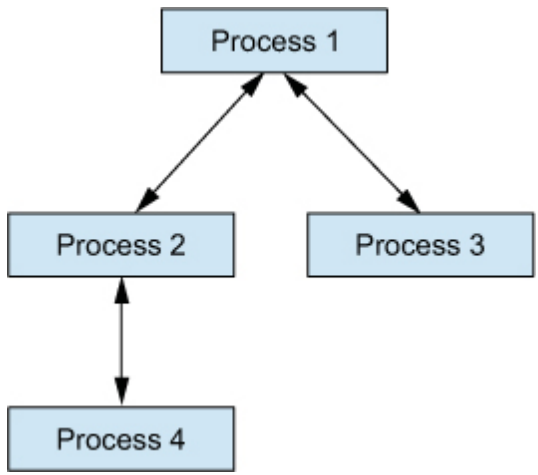
```
17:36:20.546 [error] Error in process <0.61.0>
```

Notice in particular that you don't see the output from process 2. This is because process 1 terminated abnormally, which caused an exit signal to be emitted to process 2.

One process can be linked to an arbitrary number of other processes,

and you can create as many links in the system as you want, as shown in figure 8.2. This illustrates the transitive nature of process links. In this structure, the crash of a single process will emit exit signals to all of its linked processes. If the default behavior isn't overridden, then those processes will crash as well. Ultimately, the entire tree of linked processes will be taken down.

**Figure 8.2. Example of links with multiple processes**



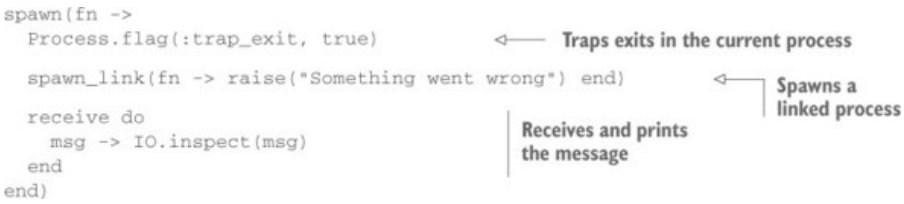

### 8.2.2. Trapping exits

You may be puzzled by the consequences of links. Earlier, I explained how process isolation makes it possible to isolate the effect of a runtime error. Links break this isolation and propagate errors over process boundaries. You can think of a link as a communication channel for providing notifications about process terminations.

Usually you don't want a linked process to crash. Instead, you want to detect the process crash and do something about it. This can be done by *trapping exits*. When a process is trapping exits, it isn't taken down when a linked process crashes. Instead, an exit signal is placed in the surviving process's message queue, in the form of a standard message. A trapping process can receive these messages and do something about the crash.

To set up an exit trap, you call `Process.flag(:trap_exit, true)`, which makes the current process trap exit signals. Let's look at

how this works:




Here, you make the parent process trap exits and then spawn a linked process that will crash. Then you receive a message and print it to the screen. The shell session produces the following output:

```
{:EXIT, #PID<0.85.0>, {%RuntimeError{message: "Something went
wrong"},
  [{:erlang, :apply, 2, []}]}}
```

The general format of the exit signal message is `{:EXIT, from_pid, exit_reason}`, where `from_pid` is the pid of the crashed process and `exit_reason` is an arbitrary term that describes the reason for process termination. If a process is terminated due to a throw or an error, the exit reason is a tuple in form `{reason, where}`, with `where` containing the stack trace. Otherwise, if a process is terminated due to an exit, the reason is a term provided to `exit/1`.

### 8.2.3. Monitors

As mentioned earlier, links are always bidirectional. Most of the time, this is exactly what you need, but in some cases unidirectional propagation of a process crash works better. Sometimes you need to connect two processes A and B in such a way that process A is notified when B terminates, but not the other way around. In such cases, you can use a *monitor*, which is something like a unidirectional link. To monitor a process, you use `Process.monitor`:

```
monitor_ref = Process.monitor(target_pid)
```

This makes the current process monitor the target process. The result is a unique reference that identifies the monitor. A single process can create multiple monitors.

If the monitored process dies, your process receives a message in the format `{:DOWN, monitor_ref, :process, from_pid, exit_reason}`. If you want to, you can also stop the monitor by calling `Process.demonitor(monitor_ref)`.

Here's a quick example:

There are two main differences between monitors and links. First, monitors are unidirectional—only the process that created a monitor receives notifications. In addition, unlike a link, the observer process won't crash when the monitored process terminates. Instead, a message is sent, which you can handle or ignore.

**Exits are propagated through gen_server calls**

When you issue a synchronous request via `GenServer.call`, if a server process crashes, then an exit signal will occur in your client process. This is a simple but very important example of cross-process error propagation. Internally, `:gen_server` sets up a temporary monitor that targets the server process. While waiting for a response from the server, if a `:DOWN` message is received, `:gen_server` can detect that a process has crashed and raise a corresponding exit signal in the client process.

Links, exit traps, and monitors make it possible to detect errors in a concurrent system. You can introduce a process whose only responsibility is to receive links and monitor notifications, and do something when a process crashes. Such processes, called *supervisors*, are the primary tool of error recovery in concurrent systems.

## 8.3. Supervisors

The idea behind supervision is simple. You have a bunch of *worker* processes that do meaningful work. Each worker is supervised by a *supervisor process*. Whenever a worker terminates, the supervisor starts another one in its place. The supervisor does nothing but supervise, which makes its code simple, error-free, and thus unlikely to crash. This pattern is so important and frequent that OTP provides the corresponding supervisor behaviour.

### 8.3.1. Supervisor behaviour

Recall from chapter 6 that a *behaviour* (using the British spelling of the word) is a generic piece of code that you can plug into. In the case of supervisors, a behaviour implemented in the generic `:supervisor` module works as follows:

1. The behaviour starts and runs the supervisor process.
2. The supervisor process traps exits.
3. From within the supervisor process, child processes are started and linked to the supervisor process.
4. If a crash happens, the supervisor process receives an exit signal and performs corrective actions, such as restarting the crashed process.
5. If a supervisor is terminated, child processes are terminated immediately.

How does the supervisor behaviour know which processes must be started? This is the task of the callback module—the one you write and use to plug into the behaviour. In the callback module, you specify which children must be started and what to do when a child crashes.

Let's see this in action by introducing a basic supervisor to the to-do

system. Figure 8.3 recaps the processes in the system:

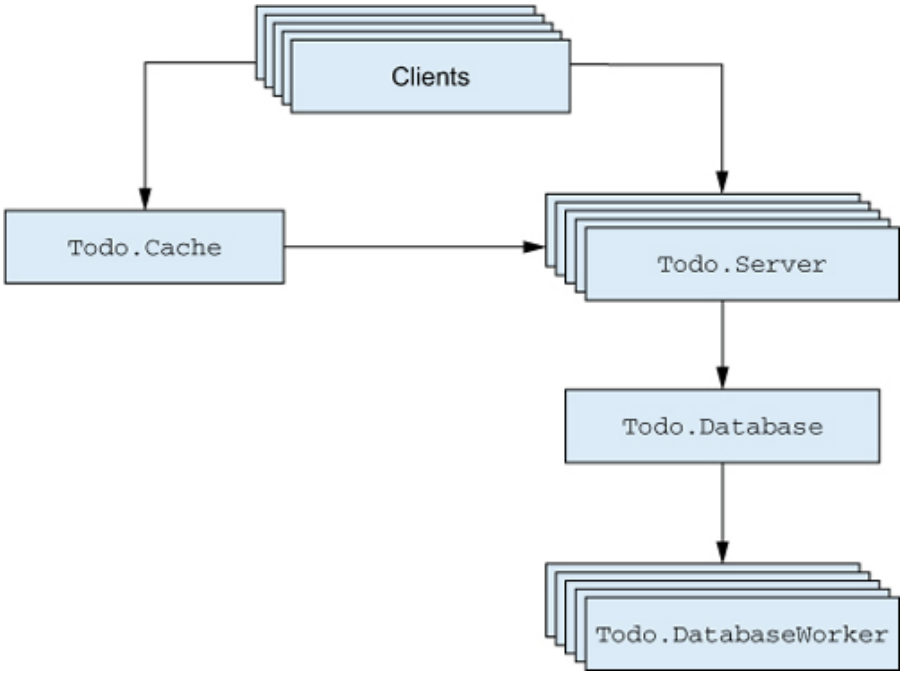
- `Todo.DatabaseWorker`—Performs read/write operations on the database (in this case, a file system).
- `Todo.Database`—Maintains a pool of database workers, and forwards database requests to them. Forwarding is done with affinity: the same item always ends up on the same worker.
- `Todo.Server`—Keeps a single instance of the `Todo.List` abstraction.
- `Todo.Cache`—Maintains a collection of to-do servers and is responsible for their creation and retrieval.

The to-do cache process is the system entry point. You use it to start the entire system, so it can be considered the root of the system. Let's start handling errors by introducing a supervisor that supervises the to-do cache.

#### Preparing the existing code

Before you start creating the supervisor, you need to make a couple of changes to the cache. First, you'll register the cache process under an alias. This will allow you to interact with the process without needing to know its pid.

**Figure 8.3. Processes in the to-do system**




You also need to create a link while starting the to-do cache process. This is required if you want to run the process under a supervisor. Why is the supervisor using links rather than monitors? Because links work in both directions, so the termination of a supervisor means all of its children will be automatically taken down. This in turn allows you to properly terminate any part of the system without leaving behind dangling processes. You'll see how this works in this chapter and the next, where you work with finer-grained supervision.

Creating a link to a supervisor is as simple as using `GenServer.start_link` in place of `GenServer.start`. While you're at it, you'll also rename the corresponding `Todo.Cache` interface function to `start_link`. The changes are illustrated in the following listing.

**Listing 8.1. Changes in the to-do cache (supervised_todo_cache/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  use GenServer

  def start_link do
    IO.puts "Starting to-do cache."

    GenServer.start_link(__MODULE__, nil, name: :todo_cache)
  end

  def server_process(todo_list_name) do
    GenServer.call(:todo_cache, {:server_process, todo_list_name})
  end

  ...
end
```

Debug message

Renamed interface function

Registers under an alias and links to the caller process

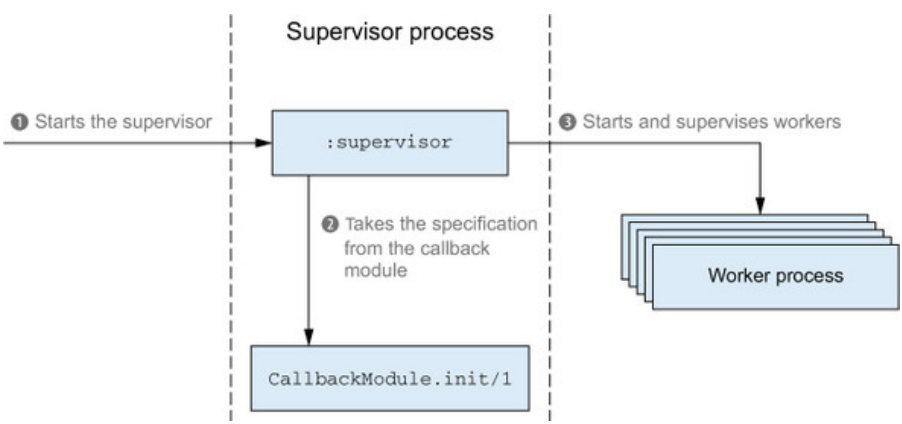
Interface function that uses the registered alias

These changes are straightforward. Notice that you also call `IO.puts/1` from the `start_link` function for debugging purposes. In fact, you can include this debug expression in all your processes.

### 8.3.2. Defining a supervisor

With these preparations in place, you can implement a simple supervisor. To do so, you must define a module and implement a callback function `init/1` that provides the *specification*—a description of the processes that are started and supervised by the supervisor process. The general workflow is depicted in figure 8.4.

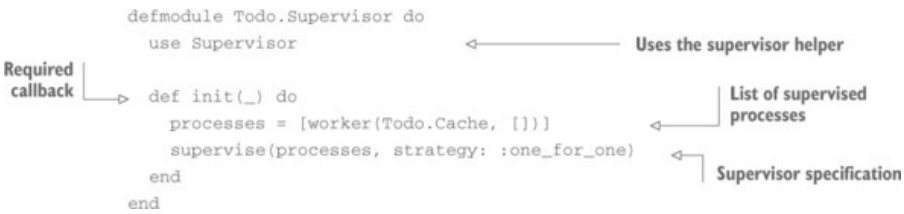
**Figure 8.4. Relationships between supervisor and worker processes**




To work with supervisors, Elixir provides a helper in the form of the `Supervisor` module (http://mng.bz/OBrI). This is best demonstrated

with code.

**Listing 8.2. To-do cache supervisor (supervised_todo_cache/lib/todo/supervisor.ex)**




The only required callback is `init/1`, which must return a supervisor specification that consists of the following:

- The list of processes that need to be started and supervised. These are often called *child processes*.
- Supervisor options, such as how to handle process termination.

In this example, the list of processes consists of a single element: a `Todo.Cache`-powered worker. The term *worker* here means "anything but a supervisor." It's common for one supervisor to supervise another; you'll see an example of this in chapter 9, when you start isolating error effects.

A worker is defined using the imported function `Supervisor.Spec.worker/2`. This function is imported to your module when you call `use Supervisor`. By invoking `worker(Todo.Cache, [])`, you're saying, "This worker must be started by calling `Todo.Cache.start_link` with an empty arguments list." The function name `start_link` is the default name, which can be overridden, but it's best to adhere to the prevalent conventions and use `start_link` whenever you start a process and link it to the caller. The result of `worker/2` is a tuple that describes the worker. Keep in mind that at this point, nothing is started; you just provide a description of child processes.

The list specifying child processes is then passed to the `supervise/2` function, which creates a tuple that describes the supervisor. The final result is returned to the supervisor behavior, which then starts the child processes, supervises them, and restarts them if they terminate.

Take special note of the `strategy: one_for_one` option. This is also known as a *restart strategy*, and it tells the supervisor behaviour how you want child termination to be handled. The `one_for_one` strategy amounts to "If a child crashes, start another one." There are a couple of other strategies (for example, "Restart all children when a single child crashes"), and we'll discuss them in chapter 9.

**Note**

The term *restart* is used casually here. Technically, a process can't be restarted. It can only be terminated; then another process, powered by the same module, can be started in its place. The new process has a different pid and doesn't share any state with the old one.

How does all of this connect? Keep in mind that a supervisor is a process, and you have to start it somehow (we'll get to this in a minute). As illustrated in figure 8.4, when you start the supervisor process, the following things happen:

1. The `init/1` callback is invoked. It must return the supervisor specification that describes how to start children and what to do in case of a crash (restart strategy).

2. Given this specification, the supervisor behaviour starts the corresponding child processes.

3. If a child crashes, the information is propagated to the supervisor (via a link mechanism), which performs a corresponding action according to the specified restart strategy. In the case of a `one_for_one` supervisor, it starts a new child in place of the old one.

### Module-less supervisors

It isn't strictly necessary to define a module that plugs into a supervisor behaviour. The `Supervisor` module provides a way to specify children when starting the supervisor. You won't be using this approach here, to avoid confusion and so you can better see how the behaviour and plug-in code connect. But when you've grasped the concept of supervisors, you may want to check the documentation for the `Supervisor` module and research how to use module-less

supervisors, which may simplify your code in some situations.

### 8.3.3. Starting a supervisor

Finally, it's time to test the whole thing. To start the supervisor process, you can use the `Supervisor.start` or `Supervisor.start_link` function. The latter is used to link the supervisor process to the current one.

Just as with `gen_server`, it's worth wrapping this into an interface function, as shown in the next listing.

**Listing 8.3. Starting the supervisor (supervised_todo_cache/lib/todo/supervisor.ex)**

```
defmodule Todo.Supervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, nil)
  end

  ...
end
```

Starts the supervisor process

Here you call the generic behaviour function `start_link/2`, sending it a callback module and an argument that will be passed to the `init/1` callback function.

Let's try it. Change the current folder to supervised_todo_cache, and start the shell (`iex -S mix`). Now you can start the supervisor:

```
iex(1)> Todo.Supervisor.start_link
Starting to-do cache.
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
{:ok, #PID<0.72.0>}
```

While starting the supervisor, you provide the callback module (the

one you just defined) and the argument to be passed to the `init/1` callback.

As soon as the supervisor process is started, the behaviour starts any child processes defined in the specification. In this case, this is just the to-do cache, which then starts the database server.

As is the case with `gen_server`, the result of `start` and `start_link` is in form of `{:ok, pid}`, with `pid` identifying the supervisor process. Notice that you don't know the pid of the child process (to-do cache). So how can you interact with it? This is the reason you registered the to-do cache under an alias. By having a registered alias, you can interact with the cache without needing to know its pid. Let's try this:

```
iex(2)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.64.0>

iex(3)> Todo.Server.entries(bobs_list, {2013, 12, 19})
[]
```

This proves that the system still works correctly. Now, let's check whether the to-do cache is restarted when something goes wrong. There are a couple of ways to test this. One approach is to call `raise/1`, `exit/1`, or `throw/1` from within `Todo.Cache.handle_call/3` (or any other callback that runs in a server process) and thus induce a process crash.

Another possibility is to manually terminate the process from outside (another process). To terminate any process, you can use the `Process.exit/2` function, which accepts a pid and the exit reason. The function sends an exit signal to the given process. You can try this, although you don't know the pid of the to-do cache process. But you do know the process alias, and this can be used to obtain the pid using the `Process.whereis/1` function:

```
iex(4)> Process.whereis(:todo_cache)
#PID<0.58.0>
```

Let's kill the process:

```
iex(5)> Process.whereis(:todo_cache) |> Process.exit(:kill)
Starting the to-do cache.
Starting database server.
```

Here, you send the cache an exit signal with the reason `:kill`. The exit reason can be any term, but `:kill` is treated in a special way—it ensures that the target process is unconditionally taken down, even if the process is trapping exits.

As you can see from the output, the process is immediately restarted. You can also prove that to-do cache is now a process with a different pid:

```
iex(6)> Process.whereis(:todo_cache)
#PID<0.69.0>
```

And of course, you can use the new process, just as you did the old one:

```
iex(7)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.70.0>
```

**Aliases allow process discovery**

It's important to explain why you register the to-do cache under a local alias. You should always keep in mind that in order to talk to a process, you need to have its pid. In chapter 7, you used a naive approach, somewhat resembling that of typical OO systems, where you created a process and then passed around its pid. This works fine until you enter the supervisor realm.

The problem is that supervised processes can be restarted. Remember that restarting boils down to starting another process in place of the old one—and the new process has a different pid. This means any reference to the pid of the crashed process becomes invalid, identifying a nonexistent process. This is why registered aliases are important. They provide a reliable way of finding a process and talking to it, regardless of possible process restarts.

Notice in the previous snippet that the server for Bob's list is started

again. When you killed the to-do cache, the old process structure was left dangling. The new cache process has its own private state and therefore uses a new set of to-do servers. Leaving dangling processes is definitely not a good thing, so let's fix it immediately.

### 8.3.4. Linking all processes

When the supervisor restarts the to-do cache, you'll get a completely separate process hierarchy, and there will be a new set of to-do server processes that are in no way related to the previous ones. The previous to-do servers will be unused garbage that are still running and consuming both memory and CPU resources.

Let's demonstrate this. First, start the system, and request one to-do server:

```
iex(1)> Todo.Supervisor.start_link
```

Now, get one to-do server:

```
iex(2)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.64.0>
```

A cached to-do server is, of course, not started on subsequent requests:

```
iex(3)> Todo.Cache.server_process("Bob's list")
#PID<0.64.0>
```

Let's check the number of running processes:

```
iex(4)> length(:erlang.processes)
43
```

Now, terminate the to-do cache:

```
iex(5)> Process.whereis(:todo_cache) |> Process.exit(:kill)
Starting to-do cache.
Starting database server.
```

Finally, get a to-do server for Bob's list:

```
iex(6)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.
#PID<0.70.0>
```

As you can see, after you restart the to-do cache, retrieving a previously fetched server creates a new process. This isn't surprising, because you killed the previous cache process, which also destroyed the process state.

When a process terminates, its state is released, and the new process starts with the fresh state. If you want to preserve the state, you must handle it yourself; we'll discuss this in chapter 9.

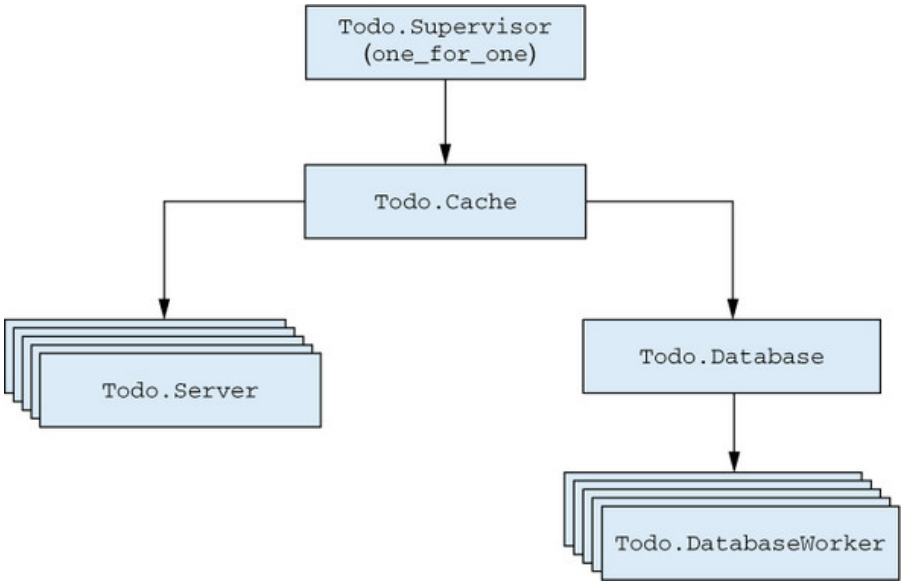
In any case, after the cache process is restarted, you have a completely new process that has no notion of what was previously cached. At the same time, your old cache structure (to-do servers) isn't cleaned up. You can see this by rechecking the number of running processes:

```
iex(7)> length(:erlang.processes)
44
```

You have one additional process, which is the previously started to-do server for Bob's list. This obviously isn't good. Terminating a to-do cache destroys its state, so you should also take down all existing to-do servers. This way, you ensure proper process termination.

To do this, you must establish links between processes. Each to-do server must be linked to the cache. Going further, you'll also link the database server to the to-do cache and the database workers to the database server. This will effectively ensure that the entire structure is linked, as illustrated in figure 8.5.

**Figure 8.5. Linking all processes in the to-do system**




This is the primary way to achieve process consistency. By linking a group of interdependent processes, you can ensure that the crash of one process takes down its dependencies as well. Regardless of which process crashes, links make sure the entire structure is terminated. Because this will lead to the termination of the cache process, it will be noticed by the supervisor, which will start a new system.

This is a proper error-recovery approach: you can detect an error in any part of the system and recover from it without leaving behind dangling processes. On the downside, you're allowing errors to have a wide impact. An error in a single database worker or a single to-do server will take down the entire structure. This is far from perfect, and you'll make improvements in chapter 9.

But for now, let's stick with this simple approach and implement the required code. In your present system, you have a to-do supervisor that starts and supervises the cache. You must ensure that the cache is directly or indirectly linked to all other worker processes.

The change is simple. All you need to do is change `start` to `start_link` for all the processes in the project. In the corresponding modules, you currently have something like this:

```
def start(...) do
```

```
  GenServer.start(...)
end
```

This snippet must be transformed into the following:

```
def start_link(...) do
  GenServer.start_link(...)
end
```

And of course, every `Module.start` call must be replaced with `Module.start_link`. These changes are mechanical, and the code isn't presented here. The complete solution resides in the todo_link folder.

Let's see how the new system works:

```
iex(1)> Todo.Supervisor.start_link

iex(2)> Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.

iex(3)> length(:erlang.processes)
43

iex(4)> Process.whereis(:todo_cache) |> Process.exit(:kill)

iex(5)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list.

iex(6)> length(:erlang.processes)
43
```

Terminates the entire process structure

The process count remains the same.

When you crash a process, the entire structure is terminated, and the new process starts in its place. Links ensure that dependent processes are terminated as well, which keeps the system consistent.

### 8.3.5. Restart frequency

It's important to keep in mind that a supervisor won't restart a child process forever. The supervisor relies on the *maximum restart frequency*, which defines how many restarts are allowed in a given time period. By default, the maximum restart frequency is five restarts in five seconds. If this frequency is exceeded, the supervisor gives up and terminates itself. When the supervisor terminates, all of its children are terminated as well (because all children are linked to the supervisor).

Let's verify this in the shell. First, start the supervisor:

```
iex(1)> Todo.Supervisor.start_link
Starting the to-do cache.
```

Now you need to perform frequent restarts of the to-do cache process:

```
iex(1)> for _ <- 1..6 do
          Process.whereis(:todo_cache) |> Process.exit(:kill)
          :timer.sleep(200)
        end
```

Here you terminate the cache process and sleep for a short while, allowing the supervisor to restart the process. This is done six times, meaning that in the last iteration, you'll exceed the default maximum restart frequency (five restarts in five seconds).

Here's the output:

```
Starting the to-do cache.
Starting database server.
...

** (EXIT from #PID<0.57.0>) :shutdown
```

Repeated five times

Supervisor gives up and terminates

After the maximum restart frequency has been exceeded, the supervisor gave up and terminated, taking down the child processes as well.

You may wonder about the reason for this mechanism. The purpose of a supervisor is to detect that a process has crashed and ensure that another one is started in its place, in hopes that this will fix the problem. If it doesn't, there's no point in infinite restarting. If too many restarts occur in a given time interval, it's clear that the problem can't be fixed. In this case, the only sensible thing a supervisor can do is to give up and terminate itself, which in addition terminates all of its children.

This mechanism plays an important role in so-called *supervision trees*, where supervisors and workers are organized in a deeper hierarchy that allows you to control how the system recovers from errors. This will be thoroughly explained in the next chapter, where you'll build a

fine-grained supervision tree.

## 8.4. Summary

In this chapter, you've seen the basic error-handling techniques in a concurrent system. A few points are worth remembering:

- There are three types of runtime errors: throws, errors, and exits.
- When a runtime error occurs, execution moves up the stack to the corresponding `try` block. If an error isn't handled, a process will crash.
- Process termination can be detected in another process. To do this, you can use links or monitors.
- Links are bidirectional—a crash of either process is propagated to the other process.
- By default, when a process terminates abnormally, all processes linked to it terminate as well. By trapping exits, you can react to the crash of a linked process and do something about it.
- Supervisors can be used to start, supervise, and restart crashed processes.

There's much more to fault tolerance than discussed in this chapter. We'll continue exploring important patterns in the next chapter, where you'll learn how to minimize the impact of errors.

# Chapter 9. Isolating error effects

*This chapter covers*

- Understanding supervision trees
- Starting workers dynamically
- “Let it crash”

In chapter 8, you learned about the basic theory behind error handling in concurrent systems based on the concept of supervisors. The idea is to have a process whose only job is to supervise other processes and to restart them if they crash. This gives you a way to deal with all sorts of unexpected errors in your system. Regardless of what goes wrong in a worker process, you can be sure that the supervisor will detect an error and restart the worker.

In addition to providing basic error detection and recovery, supervisors play an important role in isolating error effects. By placing individual workers directly under a supervisor, you can confine an error’s impact to a single worker. This has an important benefit: it makes your system more available to its clients. Unexpected errors will occur no matter how hard you try to avoid them. Isolating the effects of such errors allows other parts of the system to run and provide service while you’re recovering from the error.

For example, a database error in this book’s example to-do system shouldn’t stop the cache from working. While you’re trying to recover from whatever went wrong in the database part, you should continue to serve existing cached data, thus providing at least partial service. Going even further, an error in an individual database worker shouldn’t affect other database operations. Ultimately, if you can confine an error’s impact to a small part of the system, your system can provide most of its service all of the time.

Isolating errors and minimizing their negative effects is the topic of this chapter. The main idea is to run each worker under a supervisor, which makes it possible to restart each worker individually. You’ll see how this works in the next section, where you start to build a fine-

grained supervision tree.

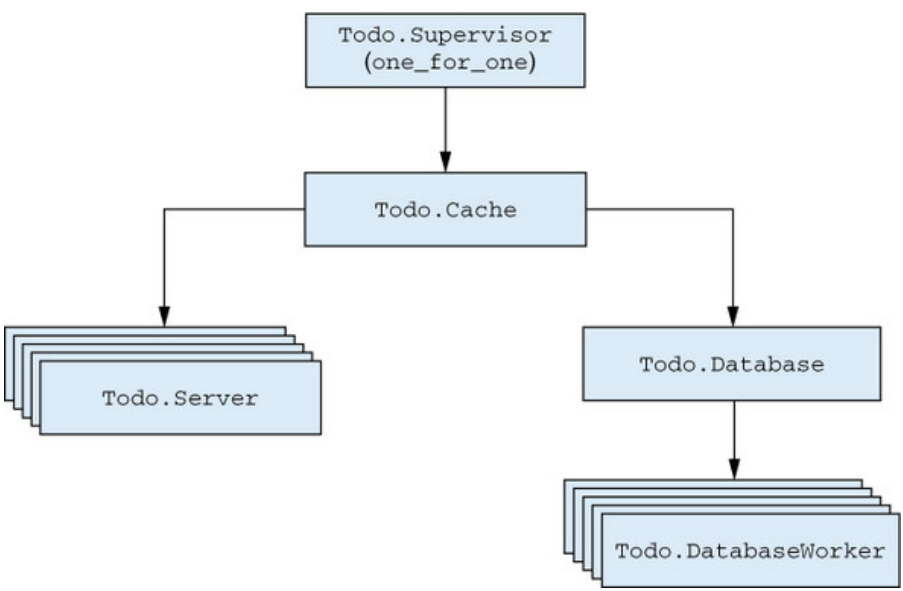
## 9.1. Supervision trees

In this section, we'll discuss how to reduce the effect of an error on the entire system. The basic tools are processes, links, and supervisors, and the general approach is fairly simple. You always have to consider what will happen to the rest of the system if a process crashes due to an error; and you should take corrective measures when an error's impact is too wide (the error affects too many processes).

### 9.1.1. Separating loosely dependent parts

Let's look at how errors are propagated in the to-do system. Links between processes are depicted in figure 9.1.

**Figure 9.1. Process links in the to-do system**




As you can see in the diagram, the entire structure is connected. Regardless of which process crashes, the exit signal will be propagated to its linked processes. Ultimately, the to-do cache process will crash

as well, and this will be noticed by the `Todo.Supervisor`, which will in turn restart the cache process.

This is a correct error-handling approach because you restart the system and don't leave behind any dangling processes. But such a recovery approach is too coarse.

Wherever an error happens, the entire system is restarted. In the case of a database error, the entire to-do cache will terminate. Similarly, an error in one to-do server process will take down all the database workers.

This coarse-grained error recovery is due to the fact that you're starting worker processes from within other workers. For example, a database server is started from the to-do cache. To reduce error effects, you need to start individual workers from the supervisor. Such a scheme makes it possible for the supervisor to supervise and restart each worker separately.

Let's see how to do this. First, you'll move the database server so it's started directly from the supervisor. This will allow you to isolate database errors from those that happen in the cache.

Placing the database server under supervision is simple enough. You must remove the call to `Todo.Database.start_link` from `Todo.Cache.init/1`. Then, you have to add another child to the supervisor specification, as illustrated in the following listing.

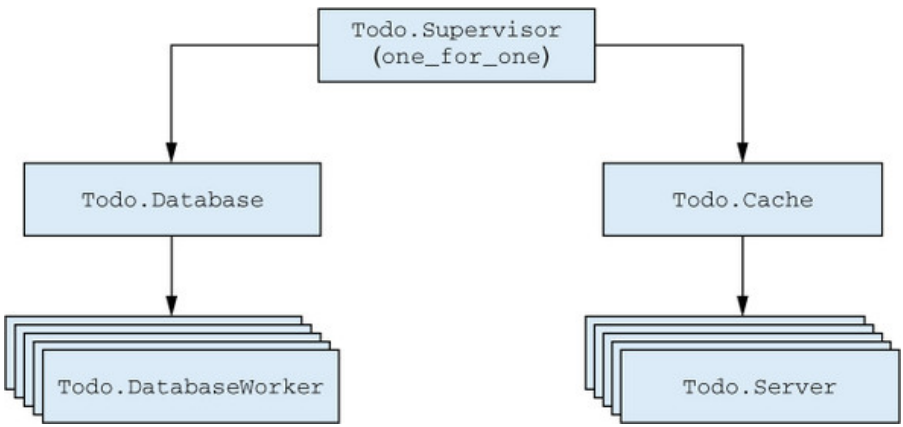
**Listing 9.1. Supervising database server (supervise_database/lib/todo/supervisor.ex)**

```
defmodule Todo.Supervisor do
  ...

  def init(_) do
    processes = [
      worker(Todo.Database, ["./persist/"]),   <---- Starts the database server
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

These changes ensure that the cache and the database are separated,

as shown in figure 9.2. Running both the database and cache processes under the supervisor makes it possible to restart each worker individually. An error in the database worker will crash the entire database structure, but the cache will remain undisturbed. This means all clients reading from the cache will be able to get their results while the database part is restarting.

**Figure 9.2. Separated supervision of the database and the cache**




Let's verify this. Go to the supervise_database folder, and start the shell (`iex -S mix`). Then start the system:

```
iex(1)> Todo.Supervisor.start_link
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
Starting to-do cache.
```

Now, kill the database server:

```
iex(2)> Process.whereis(:database_server) |> Process.exit(:kill)
Starting database server.
Starting database worker.
Starting database worker.
Starting database worker.
```

As you can see from the output, only database-related processes are restarted. The same is true if you terminate the to-do cache. By placing both processes under supervision, you localize the negative impact of an error. A cache error will have no effect on the database part, and vice versa.

Recall chapter 8's discussion of process isolation. Because each part is implemented in a separate process, the database server and the to-do cache are isolated and don't affect each other. Of course, these processes are indirectly linked via the supervisor, but the supervisor is trapping exit signals, thus preventing further propagation. This is in particular a property of `one_for_one` supervisors—they confine an error's impact to a single worker and take the corrective measure (restart) only on that process.

**Child processes are started synchronously**

In this example, the supervisor starts two child processes. It's important to be aware that children are started synchronously, in the order specified. The supervisor starts a child, waits for it to finish, and then moves on to start the next child. When the worker is a `gen_server`, the next child is started only after the `init/1` callback function for the current child is finished.

You may recall from chapter 7 that `init/1` shouldn't run for a long time. This is precisely why. If `Todo.Database` was taking, say, five minutes to start, you wouldn't have the to-do cache available all that time. Always make sure your `init/1` functions run fast, and use the trick mentioned in chapter 7 (a process that sends itself a message during initialization) when you need more complex initialization.

### 9.1.2. Rich process discovery

Although you have some basic error isolation, there's still a lot to be desired. An error in one database worker will crash the entire database structure and terminate all running database operations. Ideally, you want to confine a database error to a single worker. This means each database worker has to be directly supervised.

There is one problem with this approach. Recall that in the current version, the database server starts the workers and keeps their pids in its internal list. But if a process is started from a supervisor, you don't have access to the pid of the worker process. This is a property of the Supervisor pattern. You can't keep a process's pid for a long time, because that process might be restarted, and its successor will have a

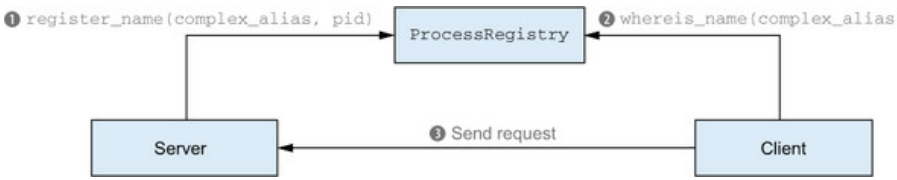
different pid.

Therefore, you need a way to give symbolic names to supervised processes and access each process via this name. When a process is restarted, the successor will register itself under the same name, which will allow you to reach the right process even after multiple restarts.

You could use registered aliases for this purpose. The problem is that aliases can only be atoms, and in this case you need something more elaborate that will allow you to use arbitrary terms, such as `{:database_worker, 1}`, `{:database_worker, 2}`, and so on. What you need is a process registry that maintains a key-value map, where the keys are aliases and the values are pids. A process registry differs from standard local registration in that aliases can be arbitrarily complex.

Every time a process is created, it can register itself to the registry under an alias. If a process is terminated and restarted, the new process will re-register itself. So, having a registry will give you a fixed point where you can discover processes (their pids). Because you need to maintain a long-running state, the registry needs to be a process. The idea is illustrated in figure 9.3.

**Figure 9.3. Discovering processes through a registry**




In step 1, the worker process registers itself during initialization. Some time later, the client process (`Todo.Server`) will query the registry for the pid of the desired worker. The client can then issue a request to the server process. Let's see a sample usage of the registry:

```
iex(1)> Todo.ProcessRegistry.start_link

iex(2)> Todo.ProcessRegistry.register_name(
          {:database_worker, 1},
          self
        )
:yes

iex(3)> Todo.ProcessRegistry.whereis_name({:database_worker, 1})
#PID<0.55.0>
```

The process registers itself.

Retrieves a process ID

This interface seems somewhat strange, with both functions having elaborate names and `register_name` returning `:yes` in the case of a successful registration (or `:no` if a process under the given alias is already registered). There is a special reason I chose this interface. By implementing exactly these kind of functions, you can combine your registry with the `gen_server` behaviour.

You'll see how this works in a minute. First you need to implement two more functions `unregister_name/1`, which unregisters the process, and `send/2`, which sends a message to the process identified via alias. Let's see how these functions can be used:

```
iex(4)> Todo.ProcessRegistry.send({:database_worker, 1}, "Hello")
iex(5)> receive do msg -> msg end
"Hello"

iex(6)> Todo.ProcessRegistry.unregister_name({:database_worker, 1})
iex(7)> Todo.ProcessRegistry.whereis_name({:database_worker, 1})
:undefined
```

Sends a message via registered alias

Unregisters the process

Notice that `Todo.ProcessRegistry` is itself a `gen_server` registered under a local alias. How can you tell? Because the process obviously maintains server-wide state (mapping of alias to pid), and none of its functions require the pid of the process registry.

Once you have a module with this interface, you can combine it with `GenServer` using *via tuples*. A via tuple must be in the form `{:via, module, alias}`. You can send such tuples to `GenServer` functions such as `start_link`, `call`, and `cast`; `GenServer` will in turn use functions from the given module to register a process, discover it, and send messages to it. Take a look at the following examples:

```
GenServer.start_link(
  Todo.DatabaseWorker,
  db_folder,
  name: {:via, Todo.ProcessRegistry, {:database_worker, 1}}
)

GenServer.call(
  {:via, Todo.ProcessRegistry, {:database_worker}, 1},
  {:get, key}
)

GenServer.cast(
  {:via, Todo.ProcessRegistry, {:database_worker}, 1},
  {:store, key, data}
)
```

Uses Todo.ProcessRegistry for alias registration

Uses Todo.ProcessRegistry for pid discovery

The implementation of the process registry is mostly a rehash of previously presented techniques. The registry is a simple `gen_server` that maintains the alias-to-pid mapping as a `HashDict`. Process discovery then amounts to querying the `HashDict` instance. The relevant `GenServer` callbacks are presented in the following listing.

**Listing 9.2. Registering a process (pool_supervision/lib/todo/process_registry.ex)**

```
defmodule Todo.ProcessRegistry do
  use GenServer

  ...

  def init(_) do
    {:ok, HashDict.new}
  end

  def handle_call({:register_name, key, pid}, _, process_registry) do
    case HashDict.get(process_registry, key) do
      nil ->
        Process.monitor(pid)
        {:reply, :yes, HashDict.put(process_registry, key, pid)}
      _ ->
        {:reply, :no, process_registry}
    end
  end


  def handle_call({:whereis_name, key}, _, process_registry) do
    {
      :reply,
      HashDict.get(process_registry, key, :undefined),
      process_registry
    }
  end

  ...
end
```

Monitors the registered process

Stores the alias-to-pid mapping

Retrieves the pid for the given alias

Notice how, during the registration, you also set up a monitor to the registered pid. This will allow you to detect the termination of a registered process and react to it by removing corresponding entries from the registry state. This is important because it lets you re-register the restarted process under the same name.

Why are you using monitors instead of links? Links aren't appropriate in this situation because a crashed registered process would take down the registry and, by extension, all other registered processes. What you need here is unidirectional crash propagation. The registry should be notified about the termination of registered processes, but not vice versa. This is what monitors are useful for.

Of course, you need to handle the `:DOWN` message, which informs you that a registered process has terminated:

```
defmodule Todo.ProcessRegistry do
  ...

  def handle_info({:DOWN, _, :process, pid, _}, process_registry) do
    {:noreply, deregister_pid(new_registry, pid)}
  end

  ...
end
```

Here you invoke the private `deregister_pid/2` function, which removes all corresponding entries from the process registry state. The implementation of this function is straightforward, so it isn't presented here.

One thing remains. To be able to work with `GenServer` and via tuples, the `Todo.ProcessRegistry` module must implement and export the `send/2` function, which must send the message to the process identified by the provided alias. But the function of the same name (`send/2`) is already auto-imported to each module from the `Kernel` module. To avoid name clashing, you need to add `import Kernel, except: [send: 2]` at the top of your module. This will autoimport the entire `Kernel` module except `send/2`. You can still use this function (and you'll need it from `send/2`) via a full qualifier: `Kernel.send/2`. The code is as follows:

```
defmodule Todo.ProcessRegistry do
  import Kernel, except: [send: 2]        <-- Disables the import of Kernel.send/2

  def send(key, message) do
    case whereis_name(key) do
      :undefined -> {:badarg, {key, message}}
      pid ->
        Kernel.send(pid, message)          <-- Uses Kernel.send/2
        pid
    end
  end

  ...
end
```

**Note**

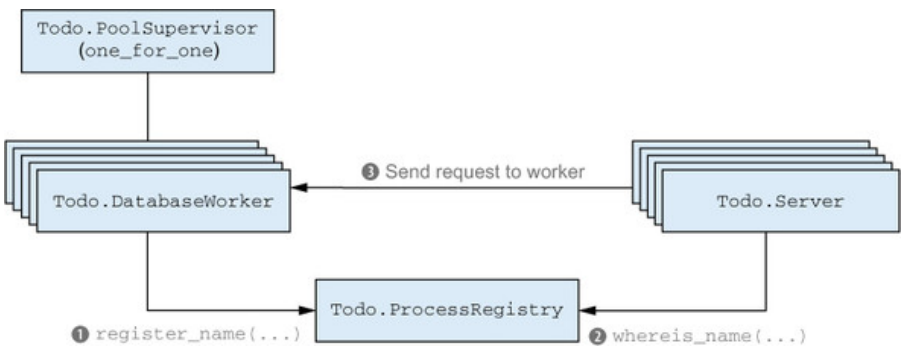
This implementation of the process registry is only for demonstration purposes. In real life, you don't need to write this yourself; a popular third-party library called `gproc` (https://github.com/uwiger/gproc) does it for you. In chapter 11, when you learn how to manage application

dependencies, you'll replace the custom implementation with `gproc`.

### 9.1.3. Supervising database workers

With the process registry in place, you can begin supervising your database workers. The general approach is depicted in figure 9.4.

**Figure 9.4. Using the process registry to discover supervised database workers**




Essentially, you must do the following:

1. Have each database worker register with the process registry under a unique alias.
2. Rely on the process registry to discover proper workers.
3. Create another supervisor that starts and supervises a pool of workers.

It's worth repeating that steps 1 and 2 (process registration and discovery) are performed internally by `GenServer`, because you'll use via tuples.

Let's begin implementing the database workers. The code is presented in the following listing.

**Listing 9.3. Registering workers (pool_supervision/lib/todo/database_worker.ex)**

```
defmodule Todo.DatabaseWorker do
  use GenServer

  def start_link(db_folder, worker_id) do
    IO.puts "Starting the database worker #{worker_id}"

    GenServer.start_link(
      __MODULE__, db_folder,
      name: via_tuple(worker_id)
    )
  end

  ...

  defp via_tuple(worker_id) do
    {:via, Todo.ProcessRegistry, {:database_worker, worker_id}}
  end
end
```

Registers via the process registry

Creates a proper via tuple

This code introduces the notion of a `worker_id`, which is an integer in the range `1..pool_size`. You accept this ID in `start_link/2` and then make a full via tuple. The alias of your worker is in the form `{:database_worker, worker_id}`, which makes it possible to distinguish database workers from other types of processes that may be registered in the process registry.

You also need to modify worker interface functions. These need worker IDs instead of pids. You'll again rely on via tuples to identify the proper process.

**Listing 9.4. Worker functions (pool_supervision/lib/todo/database_worker.ex)**

```
defmodule Todo.DatabaseWorker do
  ...

  def store(worker_id, key, data) do
    GenServer.cast(via_tuple(worker_id), {:store, key, data})
  end

  def get(worker_id, key) do
    GenServer.call(via_tuple(worker_id), {:get, key})
  end

  ...
end
```

This is a simple modification. You send via tuples instead of pids, thus relying on your own process registry to discover the process.

It's worth noting that corresponding process-registry functions (`whereis_name`) are called from the caller process (the one calling

the `store` and `get` functions). In contrast, when calling `GenServer.start_link`, `Todo.ProcessRegistry.register_name` is invoked from the process that is being registered (in this case, a database worker).

Now you can create a new supervisor that will start the pool of workers. Why are you introducing a separate supervisor? Theoretically, you could place workers under `Todo.Supervisor`, and this would work fine. But remember from the previous chapter what happens if restarts happen too often: the supervisor gives up at some point and terminates all of its children. If you keep too many children under the same supervisor, a failed restart of one child will terminate the entire system.

In this case, I made an arbitrary decision to place a distinct part of the system (the database) under a single supervisor. This approach may limit the impact of a failed restart to database operations. If restarting one database worker fails, the supervisor will terminate, which means the parent supervisor will try to restart the entire database system without touching other processes in the system. The code is presented in the next listing.

**Listing 9.5. Pool supervisor (pool_supervision/lib/todo/pool_supervisor.ex)**

```
defmodule Todo.PoolSupervisor do
  use Supervisor

  def start_link(db_folder, pool_size) do
    Supervisor.start_link(__MODULE__, {db_folder, pool_size})
  end

  def init({db_folder, pool_size}) do
    processes = for worker_id <- 1..pool_size do
      worker(
        Todo.DatabaseWorker, [db_folder, worker_id],
        id: {:database_worker, worker_id}
      )
    end

    supervise(processes, strategy: :one_for_one)
  end
end
```

Creates the list of child specifications

Supervisor child ID

This code is mostly straightforward. You create a list of child specifications and include it in a supervisor specification. Again, you use a `one_for_one` supervisor to ensure that each child is restarted individually.

Notice how you set up each worker to be started with the proper arguments. When the supervisor starts (or restarts) each worker, it calls `Todo.DatabaseWorker .start_link(db_folder, worker_id)`, passing the corresponding values. This ensures that each worker is properly started and registered.

Take special note of `id: {:database_worker, worker_id}`. This has no relation to the process-registry aliases you just used. In this context, you're creating a supervisor child ID that's used internally by the supervisor to distinguish each child. By default, if you don't provide a child ID, a worker module alias (in this case, `Todo.DatabaseWorker`) is used. But here you have multiple workers handled by the same module, so you must manually define a unique ID for each worker. Otherwise, if you try to specify multiple children with the same ID, an error will be raised during supervisor startup. In this case you use a simple tuple; its second element is the internal worker ID. This ensures that each worker has a unique supervisor child ID.

### 9.1.4. Removing the database process

With these changes in place, the database server doesn't need to know about the worker pid. Instead, it can pass the `worker_id`, which is converted to the corresponding pid via the process registry.

This approach has an important consequence: the database server doesn't need to be a process! Because the database no longer needs to maintain workers pids (this is now part of the process-registry state), there is no state to be maintained in the database server process, so you don't need the process anymore.

You'll keep the `Todo.Database` module, though. It's an interface point for clients that allows you to contain most changes in a single module, leaving the code of the clients (to-do servers) untouched.

Let's adapt the `Todo.Database` module to rely on the new process registry. First, you'll change its `start_link` function to start the pool supervisor, as demonstrated next.

**Listing 9.6. Starting the pool (pool_supervision/lib/todo/database.ex)**

```
defmodule Todo.Database do
  @pool_size 3                                    <-- Pool size

  def start_link(db_folder) do
    Todo.PoolSupervisor.start_link(db_folder, @pool_size)   <-- Starts the pool
  end

  ...
end
```

The `start_link/1` function delegates to the `Todo.PoolSupervisor` you just created. The knowledge of the pool size is defined using the module attribute `@pool_size`, and this is the only place in the code where you'll need to change it, if needed.

Finally, you can rewrite the implementation of database requests. The code is given in the following listing.

**Listing 9.7. Database requests (pool_supervision/lib/todo/database.ex)**

```
defmodule Todo.Database do
  ...

  def store(key, data) do
    key
    |> choose_worker
    |> Todo.DatabaseWorker.store(key, data)
  end

  def get(key) do
    key
    |> choose_worker
    |> Todo.DatabaseWorker.get(key)
  end

  defp choose_worker(key) do
    :erlang.phash2(key, @pool_size) + 1        <-- Chooses a worker ID
  end
end
```

Here you adapt the `store/2` and `get/1` functions to rely on the new infrastructure. Both work in a similar fashion. First you choose a worker based on the input `key`, and then you delegate to `Todo.DatabaseWorker`.

The `choose_worker/1` function takes the key, makes a numerical hash of it, normalizes it to fall in the range `0..2`, and increments by one, so that the final result is in the range `1..3`. This technique

ensures affinity—requests with the same key will end up in the same worker and thus be synchronized.

This approach seems to be different than the one depicted earlier, in figure 9.4. But keep in mind that the `store` and `get` functions are called from the server processes. Because the database is no longer a process, these invocations are delegated to `Todo.DatabaseWorker` interface functions that run in the caller process (the to-do server). Consequently, the process relationship depicted in figure 9.4 is correct. Process discovery is done from the to-do servers. The corresponding code resides in the `Todo.Database` and `Todo.DatabaseWorker` modules because you used those modules to wrap the code and make clients oblivious to message-passing details.

### 9.1.5. Starting the system

You're almost finished, but you need to make two changes in `Todo.Supervisor`. First, you must start the process registry. Additionally, you must take into account the fact that calling `Todo.Database.start_link` now starts a supervisor process. The changes are simple:

```
defmodule Todo.Supervisor do
  ...

  def init(_) do
    processes = [
      worker(Todo.ProcessRegistry, []),
      supervisor(Todo.Database, ["./persist/"]),
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

Starts the process registry

The child is a supervisor.

Take particular note how you use the `supervisor` helper function when specifying the `Todo.Database` child. By doing this, you inform the behaviour that this particular child is a supervisor. This information is mostly needed for hot code upgrades.

Keep in mind that processes are started synchronously, in the order you specify. Thus, the order in the child specification matters and isn't chosen arbitrarily. A child process always must be specified after its dependencies. In this case, you must start the registry first, because

database workers depend on it.

It's worth noting up front that this isn't proper supervision of the process registry. If a registry crashes, the restarted process won't be aware of existing registered processes. You'll deal with this problem later, in section 9.2.2.

But for now, let's see how the system works. Start everything:

```
iex(1)> Todo.Supervisor.start_link
Starting process registry
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache.
```

Now, let's verify that you can restart individual workers correctly. Because you know worker aliases are used in the process registry, you can obtain a worker's pid and terminate it:

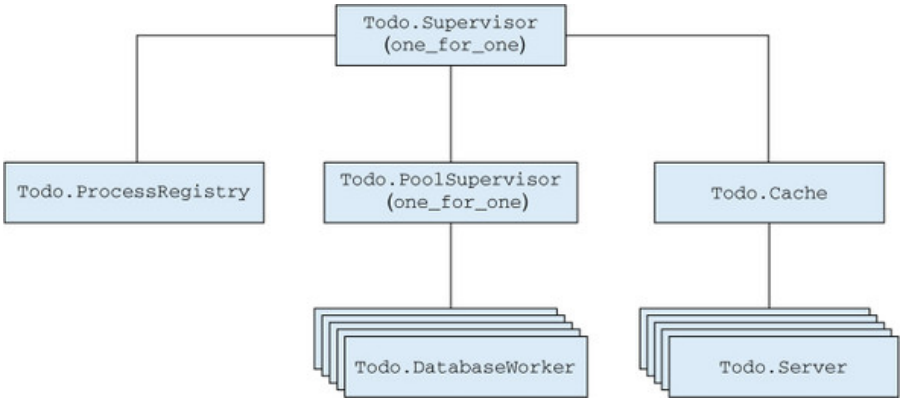
```
iex(2)> Todo.ProcessRegistry.whereis_name({:database_worker, 2}) |>
          Process.exit(:kill)
Starting database worker 2
```

The worker is restarted, as expected, and the rest of the system is undisturbed.

### 9.1.6. Organizing the supervision tree

Let's stop for a moment and reflect on what you've done so far. The relationship between processes is presented in figure 9.5.

**Figure 9.5. Supervision tree**




This is an example of a simple *supervision tree*—a nested structure of supervisors and workers. In this case, you have a top-level to-do supervisor that is responsible for starting the entire system. Underneath, it runs the process registry, the pool supervisor, and the to-do cache.

Looking at this diagram, you can reason about how errors are handled and propagated throughout the system. If a database worker crashes, the pool supervisor will restart it, leaving the rest of the system alone. If that doesn't help, you'll exceed the maximum restart frequency, and the pool supervisor will terminate all database workers and then itself.

This will be noticed by the to-do supervisor, which will then start a fresh database pool in hopes of solving the problem. What does all this restarting get you? By restarting an entire group of workers, you effectively terminate all pending database operations and begin clean. If that doesn't help, then there's nothing more you can do, so you propagate the error up the tree (in this case, killing everything). This is the point of supervision trees—you try to recover from an error locally, affecting as few processes as possible. If that doesn't work, you move up and try to restart the wider part of the system.

#### OTP-compliant processes

All processes that are started directly from a supervisor should be *OTP-compliant processes*, meaning they must comply with OTP design principles and handle OTP-specific messages. OTP behaviours, such as `gen_server` and `supervisor`, and Elixir abstractions, such as `Agent` and `Task`, can be used to run OTP-compliant processes. Plain

processes started by `spawn_link` aren't OTP compliant, so such processes shouldn't be started directly from a supervisor. You can freely start plain processes from workers such as `gen_servers`, but more often than not it's better to use OTP-compliant processes wherever possible. Doing so will improve the logging of your system, because crashes that take place in OTP-compliant processes are logged with more details.

#### Shutting down processes

An important benefit of supervision trees is the ability to stop the entire system without leaving dangling processes. When you terminate a supervisor, all of its immediate children are also terminated. If all other processes are directly or indirectly linked to those children, then they will eventually be terminated as well. Consequently, you can stop everything by terminating the top-level supervisor process.

Most often, a supervisor subtree is terminated in a controlled manner. A supervisor process will instruct its children to terminate gracefully, thus giving them the chance to do final cleanup. If some of those children are themselves supervisors, they will take down their own tree in the same way. Graceful termination of a `gen_server` worker involves invoking the `terminate/2` callback, but only if the worker process is trapping exits. Therefore, if you want to do some cleanup from a `gen_server` process, make sure you set up an exit trap from `init/1` callback.

Because graceful termination involves the possible execution of cleanup code, it may take longer than desired. An option called a *shutdown strategy* lets you control how long the supervisor will give its child to terminate gracefully. The default strategy is to wait at most five seconds for the child to terminate. If the child doesn't terminate in this time, it will be forcefully terminated. You can change this behavior by specifying `shutdown: shutdown_strategy` for each supervisor child, passing an integer indicating a time in milliseconds that's allowed for the child to terminate gracefully. Alternatively, you can pass the atom `:infinity`, which instructs the supervisor to wait indefinitely for the child to terminate. Finally, you can pass the atom `:brutal_kill`, telling the supervisor to immediately terminate the child in a forceful way. The forceful termination is done by sending a `:kill` exit signal to the process, similarly to what you did with `Process.exit(pid, :kill)`.

By default, a supervisor restarts a terminated process, regardless of the exit reason. Even if the process terminates with the reason `:normal`, it will be restarted. Sometimes you may want to alter this behavior.

For example, consider a process that handles an HTTP request or a TCP connection. If such a process fails, the socket will be closed, and there's no point in restarting the process (the remote party will be disconnected anyway). Regardless, you want to have such processes under a supervision tree, because this makes it possible to terminate the entire supervisor subtree without leaving dangling processes. In this situation, you can set up a *temporary* worker by using `worker(module, args, restart: :temporary)` in the supervisor specification. A temporary worker isn't restarted on termination.

Another option is a *transient* worker, which is restarted only if it terminates abnormally. Transient workers can be used for processes that may terminate normally, as part of the standard system workflow. For example, in the caching scheme, you use to-do server processes to keep to-do lists in memory. You may want to terminate individual servers normally if they haven't been used for a while. But if a server crashes abnormally, you want to restart it. This is exactly how transient workers function. A transient worker can be specified with `worker(module, args, restart: :transient)`.

This concludes our initial take on fine-grained supervision. You've made a number of changes that minimize the effects of errors. There's still a lot of room for improvements, and you'll continue extending the system in the next section, where you'll learn how to start workers dynamically.

## 9.2. Starting workers dynamically

The impact of a database-worker error is now confined to a single worker. It's time to do the same thing for to-do servers—the processes that manage to-do lists in memory. You'll use roughly the same

approach as you did with database workers: you'll run each to-do server under a supervisor and register the process in the process registry.

There is a twist, though. Unlike database workers, to-do servers are created dynamically, when needed. Initially, no to-do server is running; each is created on demand when you call `Todo.Cache.server_process/1`. This effectively means you can't specify supervisor children up front—you don't know how many children you need or what arguments should be passed when starting child processes.

For such cases, you need a dynamic supervisor that can start a child on demand. In OTP parlance, such supervisors have a slightly misleading name: `simple_one_for_one` supervisors.

### 9.2.1. simple_one_for_one supervisors

This is a special case of the `one_for_one` strategy, with some different properties:

- There can be multiple children, but they're all started by the same function. Hence, the child specification must contain exactly one element.
- No child is started upfront. Children can be started dynamically, on demand, by calling `Supervisor.start_child/2`.

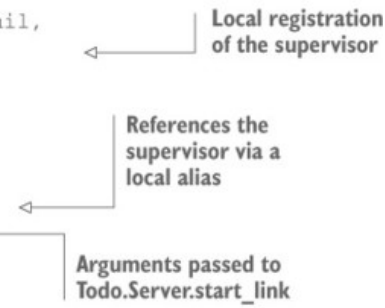
**Note**

It's possible to use other restart strategies to start children dynamically. You can, for example, use the `one_for_one` strategy and still start your children via `Supervisor.start_child/2`. But if all children are of the same type, it's more idiomatic to use `simple_one_for_one`, to indicate that you're starting children dynamically and that all children will be of the same type.

Let's create a to-do server supervisor. The implementation is provided in the following listing.

**Listing 9.8. To-do server supervisor (dynamic_workers/lib/todo/server_supervisor.ex)**

```
defmodule Todo.ServerSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, nil,
      name: :todo_server_supervisor
    )
  end

  def start_child(todo_list_name) do
    Supervisor.start_child(
      :todo_server_supervisor,
      [todo_list_name]
    )
  end

  def init(_) do
    supervise(
      [worker(Todo.Server, [])],
      strategy: :simple_one_for_one
    )
  end
end
```


Local registration of the supervisor
References the supervisor via a local alias
Arguments passed to Todo.Server.start_link
Child specification

This resembles other supervisors you've seen, with a couple of twists. First, you make the supervisor register locally under an alias, `:todo_server_supervisor`. This is needed for easier starting of children. When you want to start a child, you need to request this operation from the supervisor process. Remember, to issue a request to a specific process, you must either have its pid or know its registered alias (if it has one). To make things simple, you use an alias here.

In the `init/1` function, you still have to provide a child specification. Consider it a template that's used when starting a child. The specification of a dynamic child consists of the responsible module (`Todo.Server`) and the list of predefined, constant arguments (`[]`) that will be passed to `start_link` when starting the child.

Finally, the `start_child/1` function delegates to `Supervisor.start_child/2`. Notice how you use the `:todo_server_supervisor` alias to issue a request to the proper supervisor process.

When starting the child, you provide the list of additional arguments for `Todo.Server.start_link`. This list is appended to the one given in the child specification in `init/1`, to form the complete list of arguments passed to `Todo.Server .start_link`. The final call used to start a child thus translates to `Todo.Server .start_link(todo_list_name)`.

With this in place, you have to make the to-do server register itself to the process registry during startup. In addition, you'll implement a `whereis/1` function that returns the pid of the to-do server:

```
defmodule Todo.Server do
  use GenServer

  def start_link(name) do
    IO.puts "Starting to-do server for #{name}"
    GenServer.start_link(Todo.Server, name, name: via_tuple(name))
  end

  defp via_tuple(name) do
    {:via, Todo.ProcessRegistry, {:todo_server, name}}
  end

  def whereis(name) do
    Todo.ProcessRegistry.whereis_name({:todo_server, name})
  end

  ...
end
```

Registers in the process registry

Discovers the to-do server

With this scheme set up, you don't need to keep a mapping of to-do list names to process IDs in the to-do cache process. This mapping is now maintained in the process registry, and you can invoke `Todo.Server.whereis/1` to discover the process.

But you still need the cache process itself to prevent a special case of a race condition. Recall that when clients want to manipulate a particular to-do list, they use the `Todo.Cache.server_process/1` function to get (or create) the corresponding to-do server process. If two or more concurrent clients try to call this function at the same time for the same nonexistent list, you should create the corresponding server process only once. Therefore, you'll make a slight modification in the to-do cache implementation. In particular, `server_process/1` needs to have the following flow:

1. In the client process, check whether the to-do server process

exists. If yes, return the corresponding pid.

2. Otherwise, issue a call into the to-do cache process.

3. In the to-do cache process, recheck once again if the desired process exists. If yes, return the corresponding pid.

4. Otherwise, in the to-do cache process, create the process (via `Todo.Server-Supervisor`) and return the corresponding pid.

This ensures that multiple requests for the same nonexistent process will result in the creation of only one process, while other requests will wait for that process to be created. But processes that request an existing to-do list server won't even enter the cache process. For the sake of brevity, I omit the code, which can be found in dynamic_workers/lib/todo/cache.ex.

The final change must be made in the `Todo.Supervisor` specification, where you additionally start the new `Todo.ServerSupervisor`, as illustrated in the next listing.

**Listing 9.9. `Todo.Supervisor` (dynamic_workers/lib/todo/supervisor.ex)**

```
defmodule Todo.Supervisor do
  ...
  def init(_) do
    processes = [
      worker(Todo.ProcessRegistry, []),
      supervisor(Todo.Database, ["./persist/"]),
      supervisor(Todo.ServerSupervisor, []),     <-- New supervisor
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

With this in place, you can check if everything works. Start the shell and the entire system:

```
iex(1)> Todo.Supervisor.start_link
Starting process registry
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache
```

Now, let's get one to-do server:

```
iex(2)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list
#PID<0.65.0>
```

Repeating the request doesn't start another server:

```
iex(3)> bobs_list = Todo.Cache.server_process("Bob's list")
#PID<0.65.0>
```

In contrast, using a different to-do list name creates another process:

```
iex(4)> alices_list = Todo.Cache.server_process("Alice's list")
Starting to-do server for Alice's list
#PID<0.68.0>
```

Crash one to-do server:

```
iex(5)> Process.exit(bobs_list, :kill)
Starting to-do server for Bob's list
```

You can immediately see that Bob's to-do server has been restarted. The subsequent call to `Todo.Cache.server_process/1` will return a different pid:

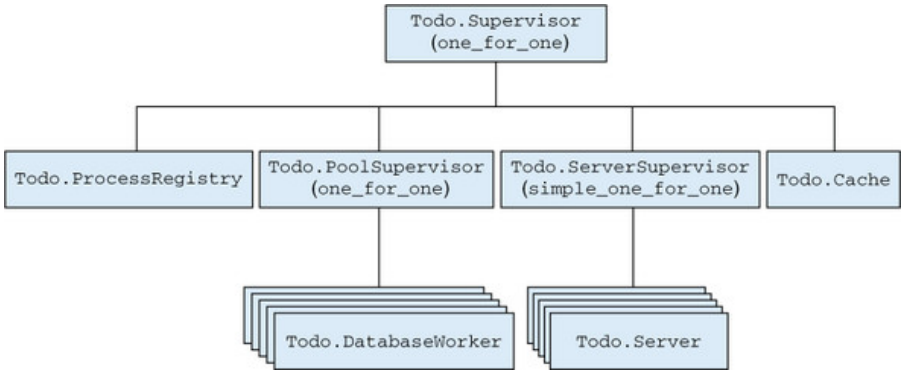
```
iex(6)> Todo.Cache.server_process("Bob's list")
#PID<0.70.0>
```

Of course, Alice's server remains undisturbed:

```
iex(7)> Todo.Cache.server_process("Alice's list")
#PID<0.68.0>
```

The supervision tree of the new code is presented in figure 9.6. The diagram clearly depicts how you supervise each process, limiting the effect of unexpected errors.

**Figure 9.6. Supervising to-do servers**




### 9.2.2. Restart strategies

So far, you've been using only `one_for_one` and `simple_one_for_one` strategies. They both handle an error by starting a new process in place of a crashed child, leaving other children alone. Occasionally, you may need a different behavior. The supervisor provides two additional restart strategies:

- `one_for_all`—When a child crashes, the supervisor terminates all other children as well and then starts all children.
- `rest_for_one`—When a child crashes, the supervisor terminates all processes from the child specification that are listed after the crashed child. Then the supervisor starts new child processes in place of the terminated ones.

Let's see this in practice. Your error handling and recovery still aren't quite correct: if a process registry is restarted, the alias-to-pid mapping is lost. Remember, the new process starts with a clean state. In this case, this is a big problem, because you can't reach existing processes via the registry.

What can you do about this? First, you must recognize that the process registry is very important for the entire system. Without it, the rest of the system can't function. Although you can afford to lose a to-do server occasionally, losing the process registry renders everything unusable. Consequently, you should keep the process-registry code as simple as possible, thus minimizing possible errors. Here, the process registry just stores data to a hash dict and reads from it. Assuming you

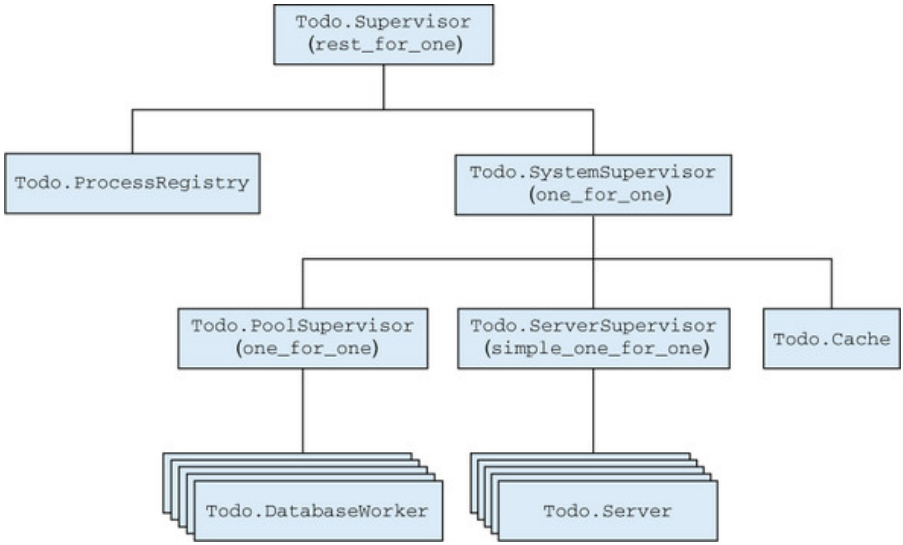
trust the `HashDict` implementation to be correct, you can safely say that the code is error free and the chance of it crashing is minimal.

Still, you should have a backup plan—a clean way to handle the case of the process registry crashing. Given that nothing else can function without the registry, your best bet is to kill everything else if the registry terminates. To do this, you need a new, top-level supervisor that will start and supervise the process registry and the rest of the to-do system. When the registry crashes, you want to terminate the rest of the system.

This is exactly the use case for the `one_for_all` and `rest_for_one` strategies, both of which terminate related processes when one process crashes. But which strategy is more appropriate for this situation? Remember, you must always ask yourself: if one process crashes, what should happen to the remaining ones? Here, when the process registry crashes, you want to kill everything else. But if the rest of the to-do system crashes, you should leave the process registry alone. This is the correct approach because the process registry doesn't depend on the rest of the system and should therefore continue to work properly even when everything else crashes.

Given the line of thinking just described, your choice should be the `rest_for_one` restart strategy. The corresponding supervision tree is shown in figure 9.7.

**Figure 9.7. `rest_for_one` supervision**




Using the `rest_for_one` strategy means a crash of the process registry will take down the to-do system as well. But when the to-do system crashes, the process registry will remain running, which is fine. Remember that you set up monitors in the process registry; when registered processes terminate, you'll unregister them, and the state of the registry will be consistent.

As an exercise, you can try to implement this supervision tree yourself. Doing so should be fairly simple, but if you get stuck, the solution can be found in the proper_registry_supervision folder.

With this, you're finished making your to-do system fault tolerant. You've introduced additional supervisor processes to the system, and you've also managed to simplify some other parts (removing the to-do cache and database server processes). You'll make many more changes to this system, but for now let's leave it and look at some important practical considerations.

## 9.3. "Let it crash"

In general, when you develop complex systems, you should employ supervisors to do your error handling and recovery. With properly

designed supervision trees, you can limit the impact of unexpected errors, and the system will hopefully recover. I can personally testify that supervisors have helped me in occasional weird situations in production, keeping the running system stable and saving me from unwanted phone calls in the middle of the night. It's also worth noting that OTP provides logging facilities, so process crashes are logged and you can see that something went wrong. It's even possible to set up an event handler that will be triggered on every process crash, thus allowing you to perform custom actions—for example, sending an email or report to an external system.

An important consequence of this style of error handling is that the `main` worker code is liberated from paranoid, defensive `try/catch` constructs. Usually these aren't needed, because you use supervisors to handle error recovery. Joe Armstrong, one of the inventors of Erlang, described such a style in his Ph.D thesis (www.erlang.org/download/armstrong_thesis_2003.pdf) as *intentional programming*. Using this approach, the code states the programmer's intention, rather than being cluttered with all sorts of defensive constructs.

This style is also known as *let it crash*. In addition to making the code shorter and more focused, *let it crash* promotes a clean-slate recovery. Remember, when a new process starts, it starts with a new state, which should be consistent. Furthermore, the message queue (mailbox) of the old process is thrown away. This will cause some requests in the system to fail. But the new process starts fresh, which gives it a better chance to resume normal operation.

*Let it crash* can initially seem confusing, and people may mistake it for the *let everything crash* approach. There are two important situations in which you should explicitly handle an error:

- Critical processes that shouldn't crash
- An error that can be dealt with in a meaningful way

### 9.3.1. Error kernel

The first case relates to what is informally called a system's *error kernel*—processes that are critical for the entire system to work and

whose state can't be restored in a simple and consistent way. Such processes are the heart of your system, and you generally don't want them to crash. In the to-do system, you could say that the process registry is your error kernel. Without the registry, other processes can't connect to each other, and the entire system stops working.

In general, you should keep the code of such important processes as simple as possible. The less logic happens in the process, the smaller the chance of a process crash. If the code of your error-kernel process is complex, consider splitting it into two processes: one that holds a state, and another that does the actual work. The former process then becomes extremely simple and thus unlikely to crash, whereas the worker process can be removed from the error kernel (because it no longer maintains a critical state).

In addition, it makes sense to include defensive `try/catch` statements in each `handle_*` callback of a critical process, thus preventing a process from crashing. Here's a simple sketch of the idea:

```
def handle_call(message, _, state) do
  try
    new_state =
      state
      |> transformation_1
      |> transformation_2
      ...

    {:reply, response, new_state}
  catch _, _ ->
    {:reply, {:error, reason}, state}
  end
end
```

Catches all errors, and uses the original state

This snippet illustrates how immutable data structures allow you to implement a fault-tolerant server. While processing a request, you make a series of transformations on the state. If anything bad happens, you use the initial state, effectively performing a rollback of all changes. This preserves state consistency while keeping the process constantly alive.

Keep in mind that this technique doesn't completely guard against a process crash. For example, you can always kill a process by invoking `Process.exit(pid, :kill)`, because a `:kill` exit reason can't be intercepted even if you're trapping exits. Therefore, you should

always have a recovery plan for a crash of a critical process. This plan amounts to setting up a proper supervision hierarchy to ensure termination of all dependent processes in case of an error-kernel process crash.

### 9.3.2. Handling expected errors

The whole point of *let it crash* is to leave recovery of unexpected errors to supervisors. But if you can predict an error and you have a way to deal with it, there's no reason to let the process crash. Here's a simple example. Look at the `:get` request in the database worker:

```
def handle_call({:get, key}, _, db_folder) do
  data = case File.read(file_name(db_folder, key)) do
    {:ok, contents} -> :erlang.binary_to_term(contents)
    _ -> nil
  end

  {:reply, data, db_folder}
end
```

Handles a file-read error

When handling a `get` request, you try to read from the file, covering the case when this reading fails. If you don't succeed, you return `nil`, treating this case as if an entry for the given key isn't in the database.

You can do better. Consider using an error only when a file isn't available. This error is identified with `{:error, :enoent}`, so the corresponding code would look like this:

```
case File.read(...) do
  {:ok, contents} -> do_something_with(contents)
  {:error, :enoent} -> nil
end
```

Notice how you rely on pattern matching here. If neither of these two expected situations happens, a pattern match will fail, and so will your process. This is the idea of *let it crash*. You deal with expected situations (the file is either available or doesn't exist), ignoring anything else that can go wrong (for example, you don't have permissions). Personally, I don't even regard this as error handling. It's a normal execution path—an expected situation that can and should be dealt with. It's certainly not something you should let crash.

In contrast, when storing data, you use `File.write!/2` (notice the exclamation), which may throw an exception and crash the process. If you didn't succeed in saving the data, then your database worker has failed, and there's no point in hiding this fact. Better to fail fast, which will cause an error that will be logged and (hopefully) noticed and fixed.

Of course, restarting may or may not help. For example, if a bug in the system allows multiple workers to write to the same file, then restarting will fix this situation—you let one process crash, and after restart, things work again. Occasional requests will fail, but the system as a whole will provide service. In a different case, perhaps you don't have proper file permissions; then restarting won't help. But the supervisor will give up and crash itself, and the system will quickly come to a halt, which is probably a good thing. No point in working if you can't persist the data.

As a general rule, if you know what to do with an error, you should definitely handle it. Otherwise, for anything unexpected, let the process crash, and ensure proper error isolation and recovery via supervisors.

### 9.3.3. Preserving the state

Keep in mind that state isn't preserved when a process is restarted. Remember from chapter 5 that a process state is its own private affair. Thus, when a process crashes, the memory it occupied is reclaimed, and the new process starts with the new state. Thus it has the important advantage of starting clean. Perhaps a process crashed due to an inconsistent state, and starting fresh may fix an error.

That said, in some cases you want the process state to survive the crash. This isn't something provided out of the box; you need to implement it yourself. The general approach is to save the state out of the process (for example, in another process or to a file) and then restore the state when the successor process is started.

You already have this functionality in the to-do server. Recall that you

have a simple database system that persists to-do lists to the disk. When the to-do server is started, the first thing it tries to do is restore the data from the database. This makes it possible for the new process to inherit the state of the old one.

In general, be careful when preserving state. As you learned in chapter 4, a typical change in a functional data abstraction goes through chained transformations:

```
new_state =
  state
  |> transformation_1(...)
  ...
  |> transformation_n(...)
```

As a rule, the state should be persisted after all transformations are completed. Only then can you be certain that your state is consistent, so this is a good opportunity to save it. For example, you do this in the to-do cache after you modify the internal data abstraction:

```
def handle_cast({:add_entry, new_entry}, {name, todo_list}) do
  new_state = TodoList.add_entry(todo_list, new_entry)
  Todo.Database.store(name, new_state)            <-- Persists the state
  {:noreply, {name, new_state}}
end
```

**Tip**

Persistent state can have a negative effect on restarts. Let's say an error is caused by a state that is somehow invalid (perhaps due to a bug). If this state is persisted, your process can never restart successfully, because the process will restore the invalid state and then crash again (either on start or when handling a request). Thus you should be careful when persisting state. If you can afford to, it's better to start clean and terminate all other dependent processes, just as you did with the process registry.

## 9.4. Summary

This concludes the topic of fine-grained supervision. You've covered a lot of ground and made your system more resilient to errors. Before continuing, here are some points worth repeating:

- Supervisors allow you to localize the impact of an error, keeping unrelated parts of the system undisturbed.
- Each process should reside somewhere in a supervision tree. This makes it possible to terminate the entire system (or an arbitrary sub-part of it) by terminating the supervisor.
- When a process crashes, its state is lost. You can deal with it by storing state outside the process, but more often than not, it's best to start with a clean state.
- In general, you should handle unexpected errors through a proper supervision hierarchy. Explicit handling through a `try` construct should be used only when you have a meaningful way to deal with an error.

After this detailed treatment of supervision trees, it's time to move on to the next topic: shared state.

# Chapter 10. Sharing state

*This chapter covers*

- Overcoming single-process bottlenecks
- Using ETS tables

From chapter 5 on, we've relied on processes to maintain state. Managing a state is one of the most important reasons to reach for processes. But when this state must be used by many processes, then the single server process (the one that manages the state) may become a bottleneck. Whenever a single process must serve many clients, those clients effectively become serialized while accessing that single process. This happens because a process can handle only one client at a time. Such behavior can have some negative consequences on the system performance and scalability. To resolve this situation, we have a facility called Erlang Term Storage (ETS) at our disposal.

## 10.1. Single-process bottleneck

Let's say you're developing a simple web server that takes a bit longer than usual to process an index request. The code in the following listing simulates this situation.

**Listing 10.1. Long processing of an index request (profile_cache/lib/web_server.ex)**

```
defmodule WebServer do
  def index do
    :timer.sleep(100)          <-- Simulates a 100ms processing time

    "<html>...</html>"
  end
end
```

This server can handle at most 10 requests per second per single core, which obviously isn't very efficient.

Assuming that the result of `index/0` doesn't change, you can cache the result in memory and serve the cached copy on subsequent requests. To keep a cache, you need a state, and therefore you must introduce a dedicated process. Here's how the cache can be used:

```
iex(1)> PageCache.start_link
{:ok, #PID<0.90.0>}

iex(2)> PageCache.cached(:index, &WebServer.index/0)
"<html>...</html>"

iex(3)> PageCache.cached(:index, &WebServer.index/0)
"<html>...</html>"
```

Computes and caches the result

Returns the cached result

This page cache allows you to send a key and a lambda that describes the computation. Internally, the cache process looks up its state and either returns a cached value for the given key or runs the provided lambda and then caches and returns the result.

**Note**

In the previous example, you capture the operation in the lambda. Here, the lambda represents a lazy operation. In fact, if the result is already present in the cache, you won't even call the lambda. You might have also noticed that you pass the lambda from one process to another. This is perfectly acceptable. When you pass a lambda to another process, all external data referenced by the lambda (closures) is deep copied.

I won't provide the implementation of the cache, because it's fairly simple and relies on familiar techniques. As an exercise, you can write it yourself or look at the existing cache in profile_cache/lib/page_cache.ex.

Let's do some quick (and inconclusive) performance measurements of this cache. Go to the folder profile_cache, run `iex -S mix`, and call the following function:

```
iex(4)> Profiler.run(PageCache, 100000)

248904 reqs/sec
```

Here, you use a custom `Profiler` module (profile_cache/lib/profiler.ex) that measures the cached execution of

`WebServer.index/0`. The call instructs your custom profiler to run 100,000 calls of `PageCache.cached/2`. The profiler measures total execution and then prints the projected throughput—the number of requests per second that you can handle.

Profiling on my machine gives approximately 250,000 requests per second, which seems decent enough. But let's check how the page cache performs when it must serve multiple clients (processes):

```
iex(5)> Profiler.run(PageCache, 100000, 100)

141728 reqs/sec
```

The third parameter to `Profiler.run` specifies the concurrency level. Here, you're starting 100 processes, each one issuing 100,000 requests to the single page-cache process. Somewhat surprisingly, the measured throughput is significantly smaller.

What happened? Keep in mind that even in moderately concurrent systems, you usually run many more processes than there are CPU cores. In this case you have 100 clients and 1 page-cache process, which is many more than the 4 CPU cores on my machine. Consequently, not all processes can run at the same time—some have to wait their turn.

As explained in chapter 5, the VM goes to great lengths to use CPUs as well as possible, but the fact remains that you have many processes competing for a limited resource. As a result, the page cache doesn't get a single CPU core all to itself. The process is often preempted while BEAM schedulers run other processes in the system. Because the page cache has fewer CPU resources for doing its job, it will take more time to compute the result.

The page cache therefore becomes a performance bottleneck and a scalability killer. As the load increases, the performance of the cache (and thus of the entire system) degrades. Even processes that don't interact with the page cache affect its performance by taking CPU time that would otherwise be allocated to the cache process.

This isn't to say that processes are bad. In general, you should strive to

run independent tasks concurrently, to improve scalability and fault tolerance. Processes should also be your first choice for maintaining changing state.

The problem isn't the many processes running in the system, but the single process on which many other processes depend. This is something you may want to avoid to promote the scalability of your system.

Another issue, not illustrated by this measure, is that no one can read from the cache while you're writing to it. Remember that you're running `WebServer.index/0` in the cached process. While you're computing the HTML for your page, you aren't able to serve anything else, even if it's cached. This could be resolved by computing the HTML out of the process and then setting the value in the cache process. But such an approach would result in excessive computations —you might end up with many duplicate executions of `WebServer.index/0` if there were many simultaneous requests for that particular page.

To resolve this, you need a global key-value store where you can safely store data and read from it, without the need to synchronize through processes. This is exactly what Erlang Term Storage (ETS) can do for you.

## 10.2. ETS tables

ETS is a separate memory-data structure where you can store Erlang terms. This makes it possible to share the system-wide state without introducing a dedicated server process. The data is kept in an *ETS table*—a dynamic memory structure where you can store tuples.

Compared to other data structures, ETS tables have some unusual characteristics:

- There's no specific ETS data type. A table is identified by its ID (a number) or a global name (an atom).

- ETS tables are mutable. A write to a table will affect subsequent read operations.
- Multiple processes can write to or read from a single ETS table. Writes and reads are concurrent.
- Minimum isolation is ensured. Multiple processes can safely write to the same row of the same table. The last write wins.
- An ETS table resides in a separate memory space. Any data coming in or out is deep copied.
- An ETS table is deeply connected to its *owner process* (by default, the process that created the table). If the owner process terminates, the ETS table is reclaimed.
- Other than on owner-process termination, there's no automatic garbage collection of an ETS table. Even if you don't hold a reference to the table, it still occupies memory.

These characteristics mean ETS tables somewhat resemble processes. In fact, it's often said that ETS tables have process semantics. You could implement ETS with processes, although such an implementation would be much less efficient. In BEAM, ETS tables are powered by C code, which ensures better speed and efficiency.

The fifth point in the previous list is especially interesting. Because data is deep-copied to and from an ETS table, there's no classical mutability problem. Once you read data from an ETS table, you have a snapshot that no one can change. Regardless of other processes possibly modifying the contents of those rows in the ETS table, the data you read remains unaffected.

### 10.2.1. Basic operations

Let's look at some examples. All functions related to ETS tables are contained in Erlang `:ets` module (www.erlang.org/doc/man/ets.html). To create a table, you can call `:ets.new/2`:

```
iex(1)> table = :ets.new(:my_table, [])
8207
```

The first argument is a table name, which is important only if you want the table to have an alias (we'll discuss this in a minute). In addition, you can pass various options, some of which are discussed

shortly. You should definitely spend some time researching the official documentation about `:ets.new/2` to see which options are possible.

The result of `:ets.new/2` is a number that represents the ETS table in the running system. The fact that it's a number is an internal detail, and you shouldn't rely on this knowledge in your code.

Because the structure is a table, you can store multiple rows into it. Each row is an arbitrarily sized tuple (with at least one element), and each tuple element can contain any Erlang term, including a deep hierarchy of nested lists, tuples, maps, or anything else you can store in a variable.

To store data, you can use `:ets.insert/2`:

```
iex(2)> :ets.insert(table, {:key_1, 1})
true
iex(3)> :ets.insert(table, {:key_2, 2})
true
iex(4)> :ets.insert(table, {:key_1, 3})
true
```

Inserts the new row

Overwrites the existing row

The first element of the tuple represents the *key*—something you can use for a fast lookup into the table. By default, ETS tables are the *set* type, which means you can't store multiple tuples with the same key. Consequently, your last write overwrites the row from the first write. To verify this, you can use `:ets_lookup`, which returns a list of rows for a given key:

```
iex(5)> :ets.lookup(table, :key_1)
[key_1: 3]

iex(6)> :ets.lookup(table, :key_2)
[key_2: 2]
```

You may wonder why the list is returned if you can have only one row per distinct key. The reason is that ETS tables support other table types, some of which allow duplicate rows. In particular, the following table types are possible:

- `:set`—Default. One row per distinct key is allowed.

- `:ordered_set`—Just like `:set`, but rows are in term order (comparison via the `<` and `>` operators)
- `:bag`—Multiple rows with the same key are allowed. But two rows can't be completely identical.
- `:duplicate_bag`—Just like `:bag`, but allows duplicate rows.

Another important option is the table's access permissions. The following values are possible:

- `:protected`—Default. The owner process can read from and write to the table. All other processes can read from the table.
- `:public`—All processes can read from and write to the table.
- `:private`—Only the owner process can access the table.

Finally, it's worth discussing the table name. This information must be an atom, and by default it serves no purpose (although, strangely enough, you must still provide it). You can create multiple tables with the same name, and they are still different tables.

But if you provide a `:named_table` option, the table becomes accessible via its name:

```
iex(1)> :ets.new(:my_table, [:named_table])
:my_table
iex(2)> :ets.insert(:my_table, {:key_1, 3})
iex(3)> :ets.lookup(:my_table, :key_2)
[]
```



In this sense, a table name resembles a locally registered process alias. It's a symbolic name of a table, and it relieves you from having to pass around the ETS numerical ID.

Of course, trying to create a duplicate named table will result in an error:

```
iex(4)> :ets.new(:my_table, [:named_table])
** (ArgumentError) argument error
    (stdlib) :ets.new(:my_table, [:named_table])
```

**Tip**

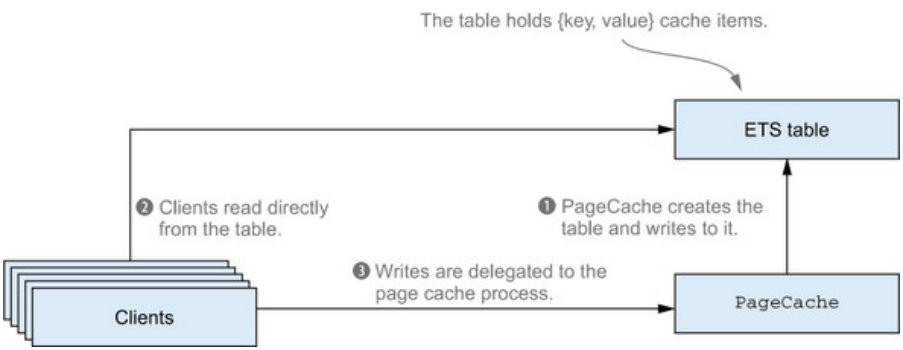
ETS tables are a limited resource. You can create at most 1,400 tables

per BEAM instance. This number can be increased by setting the environment variable `ERL_MAX_ETS_TABLES` while starting the system (for example, `iex --erl '-env ERL_MAX_ETS_TABLES 10000'`). The reason for the limitation is that an ETS table initially takes up a couple of kilobytes of memory. You should in general defer from overusing ETS tables; having a large number of them in the system is usually an indication of misuse.

### 10.2.2. ETS powered page cache

Equipped with this new knowledge, let's rewrite the page cache to rely on ETS tables. The general idea is presented in figure 10.1.

**Figure 10.1. Managing the cache in the ETS table**


The table holds {key, value} cache items.
ETS table
❷ Clients read directly from the table.
❶ PageCache creates the table and writes to it.
❸ Writes are delegated to the page cache process.
Clients
PageCache

This approach is missing some small-print details, but you'll deal with them during implementation. The most important thing to notice is that the cache is read from client processes, meaning there won't be any blocked reads. Assuming you have a high cache hit ratio (which should be the case—otherwise, why have a cache?), you should get a noticeable speed-up in cache performance. There will be no message passing between clients and the cache process, and readers won't mutually block themselves.

Let's implement this. The relevant code is contained in the same project as your initial attempt. First you need to implement starting and initializing the process, as shown in the next listing.

**Listing 10.2. Initializing an ETS powered cache (profile_cache/lib/ets_page_cache.ex)**

```
defmodule EtsPageCache do
  use GenServer

  ...

  def init(_) do
    :ets.new(:ets_page_cache, [:set, :named_table, :protected])
    {:ok, nil}
  end

  ...
end
```

Creates a table

Here you use a table of set type that is named and has a protected access. Why these parameters? The set type is reasonable, because you need a key-value cache with a single value per distinct key. Using a named table simplifies the client code, because clients don't need to know the table's ID.

Finally, you opt for protected access because you'll synchronize cache writes in the cache process. This means only client process must be able to write to the table, but all processes must have read access.

This approach means writes will still block clients. Moreover, read operations will be serialized until the cache becomes populated. But if you expect a large cache-hit rate, this shouldn't present a problem, because most clients will read the cached data. If this blocking turns out to be a problem, you can always change the access to public and perform writes from within client processes. Without knowing anything else, it's better to start with this approach, because it feels easier to reason about.

Take special note that the process doesn't maintain a state (you use `nil`). You don't need a state, because the ETS table is mutable and globally accessible. Then why do you need the cache process? There are two reasons:

- It creates and owns the table. Remember, the table is alive only while its owner process is running.
- It serializes cache writes.

Let's move on and implement the read part. The interface of the cache will be the same as in your first attempt. Hence, the only thing you need is the `cached/2` function, which accepts a key, and a lambda. If

you have something cached for the key, you'll return the value. Otherwise, you must forward the request to the cache process. The implementation is given next.

**Listing 10.3. Reading from the cache (profile_cache/lib/ets_page_cache.ex)**

```
defmodule EtsPageCache do
  ...
  def cached(key, fun) do
    read_cached(key) ||
      GenServer.call(:ets_page_cache, {:cached, key, fun})
  end

  defp read_cached(key) do
    case :ets.lookup(:ets_page_cache, key) do
      [{^key, cached}] -> cached
      _ -> nil
    end
  end

  ...
end
```

Returns a cached value, or calls into the cache process

There is a row.

Nothing is found.

The crux of the matter takes place in the `read_cached/1` function. Here, you look up the table, match the result, and return either a value or `nil`.

In `cached/2`, you use the `||` short-circuit operator to either return a cached value (if it exists) or call into the process, requesting it to compute, cache, and return the result.

The only thing remaining is the implementation of the `handle_call` callback, provided in the following listing.

**Listing 10.4. Writing into the cache (profile_cache/lib/ets_page_cache.ex)**

```
defmodule EtsPageCache do
  ...
  def handle_call({:cached, key, fun}, _, state) do
    {
      :reply,
      read_cached(key) || cache_response(key, fun),
      state
    }
  end

  defp cache_response(key, fun) do
    response = fun.()
    :ets.insert(:ets_page_cache, {key, response})
    response
  end

  ...
end
```

Attempts another cache lookup

Stores the information in the cache

This is mostly straightforward, with the small twist that you're attempting another read from the cache. You do so to remove race conditions in situations when multiple clients are simultaneously performing the first read using the same key. In this case, all clients will encounter a cache miss and issue the `cached` request to the cache process.

When this happens, you only want the first `:cached` request to invoke the operation and cache its result. Therefore, you do a safety additional cache lookup in the cached process. As a consequence of serialized writes, this lookup is isolated from other writes. While you're in the cache process, you can be certain that no one else can change the information you're reading (because all writes go through the cache process).

Let's verify that the new cache works correctly:

```
iex(1)> EtsPageCache.start_link
{:ok, #PID<0.66.0>}

iex(2)> EtsPageCache.cached(:index, &WebServer.index/0)
"<html>...</html>"

iex(3)> EtsPageCache.cached(:index, &WebServer.index/0)
"<html>...</html>"
```

The cache seems to be working. Now, let's see how it performs:

```
iex(4)> Profiler.run(EtsPageCache, 100000)
```

```
2613764 reqs/sec
```

On my machine, I obtained a projected throughput of about 2.5 million requests per second. Recalling that the initial version (plain `gen_server` with serialized reads and writes) gave a result of about 250,000 requests per second, this is a 10x increase in speed!

This looks promising, but how does it hold up against multiple clients? Let's see:

```
iex(5)> Profiler.run(EtsPageCache, 100000, 100)

10568289 reqs/sec
```

Running 100 client processes yields 4x greater throughput! This isn't surprising, given that I'm running the test on a quad-core machine.

Moreover, comparing the throughput of about 10 million requests per second to that of the plain process-page cache (~140,000 requests per second) shows an improvement of two orders of magnitude. The single-process cache starts to slow down with an increase in the total number of running processes in the system. In contrast, the ETS-based cache scales better. This is mostly due to the fact that you keep the cache lookup in the client process.

The moral of the story is to try and run independent operations in separate processes. This will ensure that available CPUs are used as much as possible and promote scalability of your system.

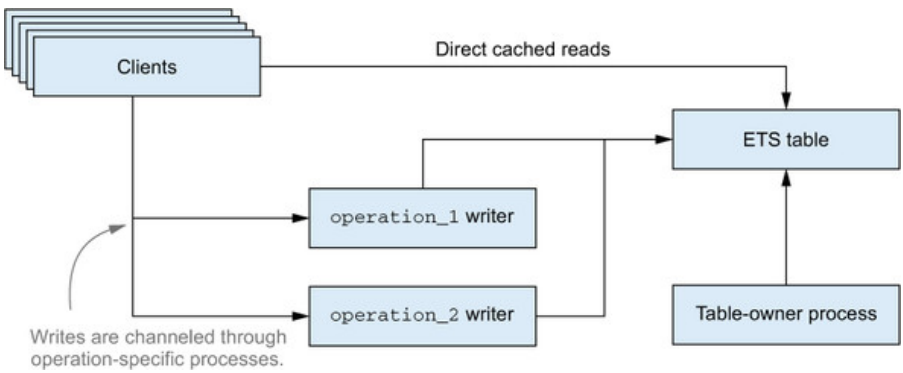
But you should try to run computations with same inputs in the single process. This allows you to cache the computation result, making sure it's executed only once. Only in the cache process can you be certain whether something was cached. Without the cache process, you might end up running multitude computations that give the same result. This in turn may increase the CPU load and slow down the entire system.

Given all this, there is some space for improvements in the current solution. Because all writes take place in the cache process, you may

face an increased error impact. If a single write operation crashes, the cache process will crash as well. This will in turn cause an error in all pending cache writes. Moreover, the entire cache will be purged, because the cache ETS table is owned by that process.

This is obviously far from perfect. What can you do about it? A simple remedy is to introduce a write process per each distinct cacheable operation. The idea is illustrated in figure 10.2.

**Figure 10.2. Finer-grained write synchronization**




In this scheme, you're synchronizing all writes of the same kind through the same process. Different kinds of computations will run in different processes, so you'll have multiple writers to the same ETS table, which is of course allowed. But because you synchronize the same kinds of writes, you won't fall into the trap of running the same computations excessively.

A nice benefit of this approach is that different types of writes aren't mutually blocking. An `operation_1` request won't block the `operation_2` request, even if both requests require a cached write. Moreover, you get finer-grained error isolation. An error in an `operation_1` request won't affect any other type of request, nor will the cache ETS table get lost.

Take special note of the table-owner process. The sole purpose of this process is to keep the table alive. Remember, an ETS table is released from memory when the owner process terminates. Therefore, you need to have a distinct, long-running process that creates and owns the table. This process usually does nothing else, which makes it

unlikely to crash. Instead of creating a dummy process that just owns a table, it's not uncommon to create the cache table in an existing supervisor. This is as simple as calling `:ets.new` from within the `init/1` supervisor callback. Keep in mind that the table needs to be named, because otherwise you can't access it from unrelated processes that reside in other parts of the supervision tree.

### 10.2.3. Other ETS operations

So far, we've covered only basic insertions and key-based lookup. These are arguably the most important operations you'll need, together with `:ets.delete/2`, which deletes all rows associated with a given key.

Key-based operations are extremely fast, and you should keep this in mind when structuring your tables. Your aim should be to maximize key-based operations, thus making ETS-related code as fast as possible.

Occasionally, you may need to perform non-key-based lookups or modifications, retrieving the list of rows based on value criteria. There are a couple of ways you can do this.

The simplest but least efficient approach is to convert the table to a list using `:ets.tab2list/1`. You can then iterate over the list and filter out your results: for example, using functions from the `Enum` and `Stream` modules.

Another option is to use `:ets.first/1` and `:ets.next/2`, which make it possible to traverse the table iteratively. Keep in mind that this traversal isn't isolated. If you want to make sure no one modifies the table while you're traversing it, you should serialize all writes and traversals in the same process. Alternatively, you can call `:ets.safe_fixtable/2`, which provides some weak guarantees about traversal. If you're iterating a fixed table, you can be certain there won't be any errors, and each element will be visited only once. But an iteration through the fixed table may or may not pick up rows that are inserted during the iteration.

Traversals and `:ets.tab2list/1` aren't very performant. Given that data is always copied from the ETS memory space to the process, you end up copying the entire table. If you only need to fetch a couple of rows based on non-key criteria, this is overkill and a waste of resources.

A better alternative is to rely on *match patterns* and *match specifications*—constructs that allow you to describe the data you want to retrieve. Consider them something like `where` filters in SQL queries.

**Match patterns**

Match patterns are a simple way to match individual rows. For example, let's say you're managing a to-do list in an ETS table:

```
iex(1)> todo_list = :ets.new(:todo_list, [:bag])
iex(2)> :ets.insert(todo_list, {{2014,5,24}, "Dentist"})
iex(3)> :ets.insert(todo_list, {{2014,5,24}, "Shopping"})
iex(4)> :ets.insert(todo_list, {{2014,5,30}, "Dentist"})
```

Here you use a `bag` ETS table, because it allows you to store multiple rows with the same key (date).

Most often, you want to query a table by key, asking, "What appointments are on the given date?"

```
iex(5)> :ets.lookup(todo_list, {2014, 5, 24})
[{{2014, 5, 24}, "Dentist"}, {{2014, 5, 24}, "Shopping"}]
```

Occasionally you may be interested in obtaining all dates for an appointment type. Here's an example of how to do this using match patterns:

```
iex(6)> :ets.match_object(todo_list, {:_, "Dentist"})
[{{2014, 5, 24}, "Dentist"}, {{2014, 5, 30}, "Dentist"}]
```

The function `:ets.match_object/2` accepts a *match pattern*—a tuple that describes the shape of the row. The atom `:_` indicates that you accept any value, so the match pattern `{:_, "Dentist"}` essentially matches all rows where the second element is `"Dentist"`.

Notice that this isn't classical pattern matching. Instead, this tuple is passed to `:ets.match_object/2`, which iterates through all rows and returns the matching ones. Therefore, when you don't care about a tuple element, you must pass an atom `:_` instead of a typical match-all anonymous variable (_). It's also worth mentioning the `:ets.match_delete/2` function, which can be used to delete multiple objects with a single statement.

In addition to being a bit more elegant, match patterns have an important performance benefit. Recall that data is always copied from the ETS table to the selected process. If you used `:ets.tab2list/1` or plain traversal, you'd have to copy every single row into your own process. In contrast, `:ets.match_object/2` performs filtering in the ETS memory space, which is more efficient.

#### Match specifications

Going further, it's possible to perform even richer queries, specifying more complex filters and even choosing individual fields you want to return. This is done by writing a full-blown *match specification* that consists of the following parts:

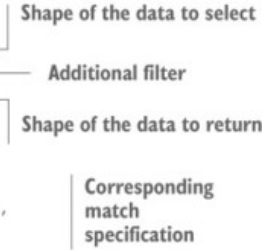
- ***Head—*** A match pattern describing the rows you want to select
- ***Guard—*** Additional filters
- ***Result—*** The shape of the returned data

Such specifications can be passed to the `:ets.select/2` function, which produces the corresponding result.

Match specifications can become complicated quickly, as you can see by looking at the documentation for `:ets.select/2` (www.erlang.org/doc/man/ets.html#select-2). Instead of writing the specifications manually, you can transform a function definition into a specification by using the `:ets.fun2ms/1` *pseudo function* (I'll explain the *pseudo* part in a bit).

For example, let's say you want to get only the dates of all appointments in the first half of 2014. The corresponding match specification can be obtained as follows:

```
iex(7)> match_spec = :ets.fun2ms(
          fn(
            {{year, month, day}, _}
          ) when year == 2014 and month <= 6 ->
            {year, month, day}
          end
        )
[{{{:"$1", :"$2", :"$3"}, :_},
  [{:andalso, {:==, :"$1", 2014}, {:"=<", :"$2", 6}}],
  [{{:"$1", :"$2", :"$3"}}]}]
```




You can now use the obtained specification in `:ets.select/2`:

```
iex(8)> :ets.select(todo_list, match_spec)
[{2014, 5, 24}, {2014, 5, 24}, {2014, 5, 30}]
```

Reaching for a full-blown match specification lets you specify a more elaborate filter. Moreover, you fetch only the fields you're interested in (appointment dates), thus avoiding needless copying of possibly large appointment titles into your memory space.

Calls to `:ets.fun2ms/1` work only at compile time, although some special support allows you to experiment with them in a shell. To use this function from your own code, you must provide a special compiler instruction:

```
defmodule MyModule do
  @compile {:parse_transform, :ms_transform}

  ...
end
```

This enables a corresponding *parse transform*, which is something like the Erlang equivalent of a macro—code that, during compilation, transforms the input code into something else. In the compiled bytecode, the input lambda you provide will be replaced with the corresponding match specification. This is why I said that `:ets.fun2ms/1` is a pseudo function. You can't invoke it at runtime, and you can't pass a dynamically created lambda.

### 10.2.4. Other use cases for ETS

Managing server-wide shared state is arguably the most common use case for ETS tables. In addition to filling a caching purpose, a globally

shared table can be used to allow processes to persist their data. Remember from chapters 8 and 9 that processes lose their state on termination. If you want to preserve the state across process restarts, the simplest way is to use a public ETS table as a means of providing in-memory state persistence. This should work reasonably fast and allow you to recover from process crashes.

But be careful about taking this road. As mentioned in chapters 8 and 9, it's generally better to recover from a crash with a clean state. You should also consider whether you can restore the state based on the data from other processes. Persisting a state in the ETS table (or anywhere else, for that matter) should be used mostly for critical processes that are part of your error kernel.

It's also possible to use ETS tables as a faster alternative to `HashDict` and maps. Given that the implementation is in C, ETS lookups should run faster than the corresponding Elixir code. There is a caveat, though. Remember that data is copied between an ETS table and a client process. Consequently, if your row data is complex and large, ETS tables may yield worse performance than pure, immutable data structures. Another important downside of ETS tables is that, unlike plain data, they can't be sent over the network to another BEAM instance. So, relying on ETS makes it harder to take advantage of distribution facilities (described in chapter 12).

In general, you should avoid using ETS and instead favor immutable structures as much as possible. Resort to ETS only in cases where you can obtain significant performance gains.

### 10.2.5. Beyond ETS

Erlang ships with two facilities that are closely related to ETS and provide a simple way of implementing an embedded database that runs in the BEAM OS process. I won't discuss these features in detail in this book, but they deserve a brief mention so you can be aware that they exist and research them more deeply on your own.

The first feature, *disk-based ETS* (DETS, www.erlang.org/doc/man/

dets.html) is disk-based term storage. Just like ETS, DETS relies on the concept of tables, and each table is managed in a single file. The interface of the corresponding `:dets` module is somewhat similar to ETS, although more limited in features. DETS provides a simple way of persisting data to the disk. Basic isolation is supported—concurrent writes are allowed, even when you're storing to the same row.

Built on top of ETS and DETS, Erlang ships with a database called Mnesia (www.erlang.org/doc/apps/mnesia/users_guide.html) that has many interesting features:

- Mnesia is an embedded database—it runs in the same BEAM instance as the rest of your Elixir/Erlang code.
- Data consists of Erlang terms.
- Tables can be in memory (powered by ETS) or disk based (powered by DETS).
- Some typical database features are provided, such as complex transactions, dirty operations, and fast lookups via secondary indices.
- Sharding and replication are supported.

These features make Mnesia a compelling option for storing data. You initialize the database from your startup Elixir/Erlang code, and you're good to go. This has the huge benefit of allowing you to run the entire system in a single OS process.

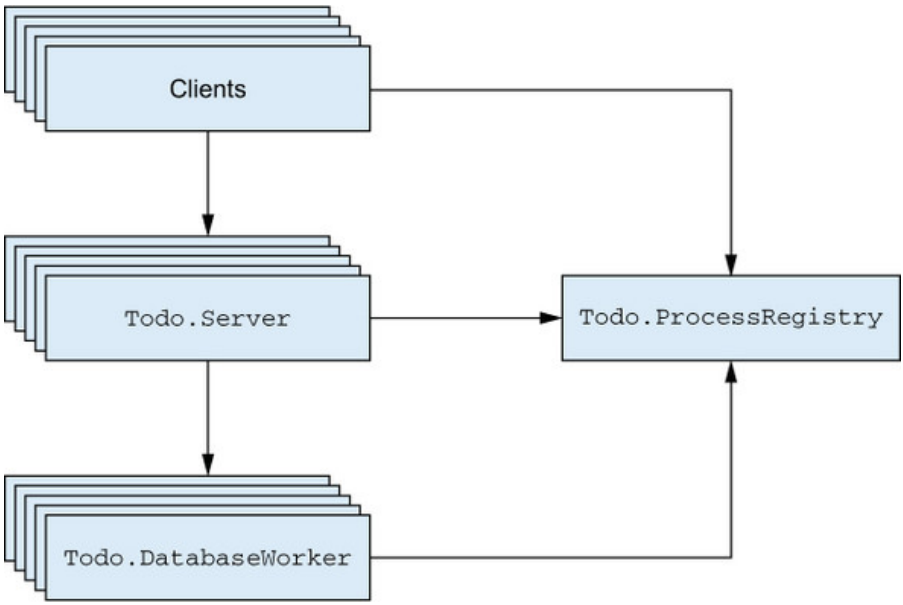
On the downside, Mnesia is a somewhat esoteric database and isn't used much outside the Elixir/Erlang community. This means there is less community and tooling support compared to popular DBMS solutions. It also takes some trickery to make Mnesia work on a larger scale. For example, one problem is that disk-based tables can't exceed 4 GB (this is a limitation of the underlying DETS storage), which means you have to fragment larger tables.

### 10.2.6. Exercise: ETS-powered process registry

It's time to apply your new knowledge to the to-do system. Recall from chapter 9 that you introduced a process registry—a place where you register processes under arbitrarily complex aliases and can look

up their pids. The current implementation of this registry is a potential bottleneck in the system, as you can see in figure 10.3.

**Figure 10.3. The process registry is a bottleneck.**




Your task is to minimize this bottleneck by allowing concurrent registry lookups. In particular, you should take the last version of your system from chapter 9 and do following:

1. Modify `Todo.ProcessRegistry` to rely on ETS tables. Use registered names as keys and pids as values.

2. Keep lookups out of the process.

3. Keep registrations (and deregistrations) in the process (to avoid race conditions).

4. When the registered process terminates and is removed from the registry, you should delete it from the cache. This will require deleting all rows for a given value (the pid of the crashed process). To do this, use `:ets.match_delete/2`, providing the corresponding simple match pattern.

The interface of the module should remain the same (which means

you only need to touch the `Todo.ProcessRegistry` module). Keep the rest of the module's behavior (monitoring for termination of registered processes and verification of duplicate registrations). If you get stuck, the solution is available in the ets_registry folder (only the process_registry.ex file has changed).

It's important to note that the resulting system will retain interprocess dependencies, as depicted in figure 10.3. In other words, the process registry will remain a bottleneck, but only for writes. Assuming that existing processes are looked up much more often than they're created, this bottleneck won't be a significant issue, and your system should be more scalable. That's the trade-off when designing a concurrent system. Occasionally you need to synchronize some actions, but the point is to avoid this synchronization where possible.

## 10.3. Summary

In this chapter, you've learned about ETS tables. Some points are worth repeating:

- An ETS table is a mutable data structure where you can store tuples.
- ETS tables reside in a separate memory space and can be accessed concurrently by many processes.
- Key-based lookups in ETS tables are very fast.
- For non-key operations, try to use match patterns or match specifications.
- When sharing data among many processes, consider using ETS for better performance and scalability. Be careful about concurrent writes, and try to serialize writes to the same table (or to the same row) through the single process.

At this point, we're finished discussing ETS tables and exploring Erlang concurrency. In the next chapter, we'll turn our attention toward building proper components called OTP applications.

# Part 3. Production

This part of the book focuses on production aspects of BEAM-powered systems. Chapter 11 discusses OTP applications. You'll learn how to package components, reuse third-party libraries, and build a simple web server. Then, in chapter 12, we look at the basic building blocks of distributed systems. Chapter 13 finishes the book by explaining how to build a deployable standalone release and how to interact with the running system.

# Chapter 11. Working with components

*This chapter covers*

- Creating OTP applications
- Working with dependencies
- Building a web server
- Managing the application environment

At this point, it's time to turn your attention toward producing releasable systems that can be deployed and started in a simple manner. To reach that goal, you need to learn about *OTP applications*—a facility that lets you organize your system into reusable components. Applications are a standard way of building Elixir/Erlang production systems and libraries, and relying on them brings various benefits, such as dependency management; simplified system starting; and the ability to build standalone, deployable releases.

In this chapter, you'll learn how to create applications and work with dependencies. In the process, you'll turn your to-do system into a proper OTP application and use some third-party libraries from the Erlang and Elixir ecosystem to expose an HTTP interface for your existing system. There's a lot of work ahead, so let's get started with OTP applications.

## 11.1. OTP applications

An OTP application is a component that consists of multiple modules and can depend on other applications. This makes it possible to start the entire system and dependent components with a single function call. As you're about to see, it's reasonably easy to turn a system into an application. Your current version of the to-do system is already an OTP application, but there are some minor details you can improve. You'll see this in action shortly; first let's look at what OTP applications consist of.

### 11.1.1. Describing an application

An application is a runtime OTP-specific construct. The resource that defines an application is called an *application resource file*—a plain-text file written in Erlang terms that describes the application (don't worry, though, you won't need to write this directly; you'll instead rely on the `mix` tool to do this for you). This file contains several pieces of information:

- The application name and version, and a description
- A list of application modules
- A list of application dependencies (which must be applications themselves)
- An optional application-callback module

Once you have this file in place, you can use Erlang's `:application` module or Elixir's `Application` module to start and stop the application. Regardless of which module you use, the underlying OTP code will dynamically load this resource file (which obviously must be somewhere in the load path) and start your application. Starting the application amounts to starting all dependencies and then the application itself, which you do by calling the callback module's `start/2` function.

### 11.1.2. Creating applications with the mix tool

Relying on the `mix` tool simplifies and automates some of the work of generating application resource files. For example, the application resource file must contain a list of all application modules. When you use `mix`, this list is generated for you automatically based on the source code and the modules you define.

Some things, such as the application name, version, and description, must of course be provided by you. The `mix` tool can then use this data and your source code to generate the corresponding resource file while compiling the project.

Let's see this in practice. Go to a temporary folder, and run `mix new`

`hello_world --sup`. This command creates a hello_world folder with the minimum `mix` project skeleton. The parameter `--sup` makes the `mix` tool generate the application callback module and start the empty (childless) supervisor from it.

You can now change to the hello_world folder and start the system with the familiar `iex -S mix`. On the surface, nothing spectacular happens. But `mix` automatically starts your application, which you can verify by calling `:application.which_applications/0`:

```
iex(1)> :application.which_applications

[{:hello_world, 'hello_world', '0.0.1'},      <-- The application is running.
 {:mix, 'mix', '1.0.0'},
 {:logger, 'logger', '1.0.0'},
 {:iex, 'iex', '1.0.0'},
 {:elixir, 'elixir', '1.0.0'},
 {:syntax_tools, 'Syntax tools', '1.6.14'},
 {:compiler, 'ERTS  CXC 138 10', '5.0'},
 {:crypto, 'CRYPTO', '3.3'},
 {:stdlib, 'ERTS  CXC 138 10', '2.0'},
 {:kernel, 'ERTS  CXC 138 10', '3.0'}]
```

As you can see, the `hello_world` application is running, together with some additional applications such as Elixir's `mix`, `iex`, and `elixir` and Erlang's `stdlib` and `kernel`.

You'll see the benefits of this shortly, but first let's look at how the application is described. The main place where you specify an application is in the mix.exs file. Here are the full contents of the generated file (comments are stripped out):

```
defmodule HelloWorld.Mixfile do
  use Mix.Project

  def project do
    [app: :hello_world,
     version: "0.0.1",
     elixir: "~> 1.0.0"
     deps: deps]
  end

  def application do
    [applications: [:logger],
     mod: {HelloWorld, []}]
  end

  defp deps do
    []
  end
end
```

Describes the project

Describes the application

Lists dependencies

The first interesting thing happens in the `project/0` function, where you describe the `mix` project. Notice `app: :hello_world`: this gives a name to your application. Only an atom is allowed as an application name, and you can use this atom to start/stop the application at runtime.

The application is described in the `application/0` function. Here you specify some options that will eventually make it to the application resource file. In this case, the description includes the list of other applications you depend on, together with the module that will be used to start the application. By default, Elixir's `logger` application (http://elixir-lang.org/docs/stable/logger/) is listed as a dependency.

Remember that starting the application amounts to calling the `start/2` function from the callback module. Consequently, when the `:hello_world` application is started, the `HelloWorld.start/2` function is invoked, and its task is to start the top-level process of the application.

Take special note of the `deps` function. This function is called from `project/0`. As its name suggests, `deps` should return a list of dependencies. You may be confused, because I just mentioned that

dependencies are listed in the `application/0` function.

What's the difference? The dependencies you return from `deps/0` are build-time dependencies. This is roughly a list of mix projects and their locations (for example, in source control or on a disk). This information is used by the `mix` tool to fetch the dependencies' source code, compile them together with your project, and create corresponding .beam files that are available when you start the system. In contrast, the dependencies returned by `application/0` are runtime dependencies—a list of OTP applications that need to be started before your own application is started.

Why is this distinction needed? Because you may have compile-time dependencies that aren't required at runtime (such as a documentation-building tool or an automated test helper). It's also possible to have runtime dependencies that aren't required at build-time, such as the `logger` application that ships with Elixir and is already compiled and available somewhere in load paths.

**Note**

It's crucial to be aware that an application is a runtime construct: a resource file that is dynamically interpreted by the corresponding OTP-specific code. When using `mix`, you describe some aspects of this file; others are derived from your code. But the application itself has meaning only at runtime. In contrast, a `mix` project is a compile-time construct. In the mix.exs file, you describe your application and implement modules. The compilation process then produces an application resource file.

### 11.1.3. The application behaviour

The critical part of the application description is `mod: {HelloWorld, []}`, specified by `application/0`. This part describes the module that will be used to start the application. When you start the application, the function `HelloWorld.start/2` is called with arguments provided in the second element of the tuple (`[]` in this case).

Obviously, you need to implement the `HelloWorld` module. This is done for you by the `mix` tool, so let's see what it looks like:

```
defmodule HelloWorld do
  use Application                          <-- Uses the Application helper

  def start(_type, _args) do               <-- Callback function
    import Supervisor.Spec, warn: false

    children = [
    ]
    opts = [strategy: :one_for_one, name: HelloWorld.Supervisor]
    Supervisor.start_link(children, opts)
  end                                      <-- Starts the top-level supervisor
end
```

An application is an OTP behaviour. There is a generic `:application` module (www.erlang.org/doc/apps/kernel/application.html) that you can use to work with applications, and there are also some Elixir helper functions in the `Application` module (http://elixir-lang.org/docs/stable/elixir/Application.html). To be able to work with these modules, you must implement your own callback module and define some callback functions.

At minimum, your callback module must contain a `start/2` function. The arguments passed are the application start type (which you'll usually ignore) and start arguments (a term specified in `mix.exs` under the `mod` key).

The task of the `start/2` callback is to start the top-level process of your system, which should usually be a supervisor. The code generated by the `mix` tool uses the generic `Supervisor` module for this, but you can also use your own dedicated supervisor module, like the ones you implemented in chapters 8 and 9.

Either way, the `start/2` callback starts the top-level process (which in turn may start other processes) and returns its result in the form `{:ok, pid}` (or `{:error, reason}` if something went wrong) to the generic `:application` code.

### 11.1.4. Starting the application

To start the application in the running BEAM instance, you can call

`Application .start/1`. This function first looks for the application resource file (which is generated by `mix`) and interprets its contents. Then it verifies whether all applications you're depending on are started. Finally, the application is started by calling the callback module's `start/2` function. There is also an `Application.ensure_ all_started/1` function available, which recursively starts all dependencies that aren't yet started.

But in general, you won't need these functions, because `mix` automatically starts the application implemented by the project. Calling `iex -S mix` automatically starts the application together with its dependencies.

If you want to start the system without the `iex` shell, you can run `mix run --no-halt` from the command line. This starts the BEAM instance but not the shell. The `--no-halt` option instruct the `mix` tool to keep the BEAM instance running forever (even if your application stops for some reason). Without this option, the BEAM instance would terminate as soon as `HelloWorld.start` finished, which happens immediately after the top-level process is started.

Another important point is that you can't start multiple instances of a single application. In this sense, an application is like a singleton in a single BEAM instance. Trying to start the already-running application will return an error:

```
$ iex -S mix

iex(1)> Application.start(:hello_world)
{:error, {:already_started, :hello_world}}
```

It's also possible to stop the application using `Application.stop/1`:

```
iex(2)> Application.stop(:hello_world)
:ok

22:28:52.029 [info] Application hello_world exited: :stopped
```

Stopping the application terminates its top-level process. This plays nicely with the supervision trees you saw in chapter 9. If all of your processes run in a supervision tree, and if your top-level process is a

supervisor, then by terminating the top-level process you also terminate all other processes created by your application, leaving no dangling processes behind. But `stop/1` stops only the specified application, leaving dependencies (other applications) running.

### 11.1.5. Library applications

You don't need to provide the `mod: ...` option from the `application/0` function in `mix.exs`:

```
...

def application do
  []
end

...
```

In this case, there's no application callback module, which in turn means there's no top-level process to be started. Oddly enough, this is still a proper OTP application. You can even start it and stop it.

What's the purpose of such applications? This technique is used for *library applications*: components that don't need to create their own supervision tree. As the name indicates, these are usually simpler libraries; a typical example is a JSON parser. For example, Erlang's own `stdlib` application (www.erlang.org/doc/apps/stdlib/index.html) is also a pure library application because it exposes various utility modules but doesn't need to manage its own supervision tree.

Library applications are useful because you can list them as runtime dependencies. This plays an important role when you start to assemble the deployable release, as you'll see in chapter 13.

### 11.1.6. Creating a to-do application

Equipped with this knowledge, let's turn your to-do system into a proper application. As mentioned earlier, it's already an OTP application, albeit a library one that doesn't implement a callback

module for the application behaviour.

Given that the to-do system runs a set of its own processes under a supervision tree, it makes sense to turn it into a full-blown application. This yields the immediate benefit of simplified system startup. Once you properly implement the application callback module, the system can be automatically started as soon as you run `iex -S mix` (or `mix run --no-halt`).

The first thing you need is the mix.exs file. Recall that you created your initial project back in chapter 7 using the `mix` tool. Therefore, you already have this file in place and only need to add some information. The complete code for mix.exs is provided in the following listing.

**Listing 11.1. Specifying application parameters (todo_app/mix.exs)**

```
defmodule Todo.Mixfile do
  use Mix.Project

  def project do
    [ app: :todo,
      version: "0.0.1",
      elixir: "~> 1.0.0",
      deps: deps
    ]
  end

  def application do
    [
      applications: [:logger],
      mod: {Todo.Application, []}
    ]
  end

  defp deps do
    [{:meck, "0.8.2", only: :test}]
  end
end
```

Application and version info

Application module and run-time dependencies

Compile-time dependencies

There is nothing new here except the implementation of the `deps` function. We'll deal with compile-time dependencies a bit later; for the moment it's enough to mention that this configures a dependency on the mocking library, which is used in tests of the sample code. The

`only: :test` option ensures that this dependency is used only when running tests with `mix test`. Therefore, the meck application isn't available to the code when you start it with `iex -S mix` (or `mix run --no-halt`).

Next, you need to implement the application callback module. This is now a simple task, as shown next.

**Listing 11.2. Implementing the application callback (todo_app/lib/todo/application.ex)**

```
defmodule Todo.Application do
  use Application

  def start(_, _) do
    Todo.Supervisor.start_link
  end
end
```

As already mentioned, starting the application is as simple as starting the top-level supervisor. Given that you've already structured your system to reside under `Todo.Supervisor` this is all it takes to turn your system into a full-blown OTP application.

Let's see it in action:

```
$ iex -S mix

Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:8:8]
[async-threads:10] [kernel-poll:false]

$ iex -S mix

Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:8:8]
[async-threads:10] [kernel-poll:false]
Starting process registry
Starting database worker 1
Starting database worker 2
Starting database worker 3
Starting to-do cache

Interactive Elixir (1.0.0) - press Ctrl+C to exit
(type h() ENTER for help)

iex(1)>
```

**The system is started automatically.**

By turning the system into an OTP application, you've made it possible to start the system automatically.

### 11.1.7. The application folder structure

Let's briefly discuss your compiled application's folder structure. Owing to the `mix` tool, most of the time you don't need to worry about this, but it can sometimes be useful to understand the folder structure of a compiled system.

#### The mix project environment

Before we look at the application structure, you should know a bit about `mix` project environments. A *project environment* is a compile-time construct, essentially an atom that can be used to vary compile-time settings.

For example, recall the `meck` dependency specified in mix.exs:

```
...

defp deps do
  [{:meck, "0.8.2", only: :test}]
end

...
```

The `only: :test` option ensures that this dependency is used only in the `test` project environment.

By default, the `mix` project environment is called `dev`, indicating that you're dealing with development. Consequently, when you run `mix compile` or `iex -S mix` (which implicitly runs the compilation), you aren't in the test environment and therefore don't depend on the `meck` project. But when you run tests with `mix test`, the project environment is automatically set to `test`, and modules from the `meck` project are available.

Moreover, `meck` isn't an application dependency. You require corresponding compiled code to be available to you in the `test` environment, but you don't require that any application be started.

You can specify a project environment by setting `MIX_ENV` OS

environment variable. By informal convention when building for production, you should use `prod` environment settings, which can be as simple as running `MIX_ENV=prod mix compile` from the shell. This is needed mostly for libraries that may produce extra code (for example, debug statements, or automatic scanning and reloading of recompiled binaries) in the `:dev` environment but omit such code otherwise.

#### The compiled code structure

Once you compile your project, compiled binaries reside in the _build/ProjectEnv folder, where ProjectEnv is the `mix` project environment that was in effect during compilation.

Because `dev` is the default environment, if you run `mix compile` or `iex -S mix`, you get binaries in the _build/dev folder. The OTP itself the following folder convention:

```
lib/
  App1/
    ebin/
    priv/

  App2/
    ebin/
    priv/

  ...
```

Here, App1 and App2 represent application names (such as `todo`). The ebin folder contains compiled binaries (.beam files and application resource files), whereas the priv folder contains application-specific private files (images, compiled C binaries, and so on). This isn't a mandatory structure, but it's a prevalent convention used in most Elixir/Erlang projects, including Elixir and Erlang. Some tools may rely on this structure, so it's best to follow this convention.

Luckily, you don't need to maintain this structure yourself, because `mix` does so automatically. The final folder structure of a compiled `mix` project has the following shape:

```
YourProjectFolder
  _build
    dev
      lib
```

```
App1
  ebin
  priv
App2
...
```

In addition to your application, the lib folder contains your compile-time dependencies. Other runtime dependencies (such as Elixir/Erlang standard applications) already reside someplace else on the disk and are accessible via the load path.

As mentioned, the application resource file resides in lib/YourApp/ebin and is named as YourApp.app. For the to-do system, the file resides in _build/dev/lib/todo/ebin/ (relative to the root project folder). When you attempt to start the application, the generic application behaviour looks for the resource file in the load paths (the same paths that are searched for compiled binaries).

This concludes our discussion of the basics of applications. Now that you have a grasp of some theory, let's look at how to work with dependencies.

### Deployable systems

Applications play an important role in building a deployable system. You'll learn about this in chapter 13, where I'll discuss OTP releases. To make a long story short, the general idea is to assemble a minimal self-contained system that consists only of required applications and the Erlang runtime. For this to work, you must turn your code into an OTP application, because only then can you specify dependencies to other applications. This is ultimately used to bundle all required applications into a single deployable release.

Therefore, anything reusable in Elixir/Erlang should reside in an application. This holds even for Elixir/Erlang: examples include the `elixir` application, which bundles the Elixir standard library, as well as `iex` and `mix`, which are implemented as separate applications. The same thing happens in Erlang, which is divided into many applications (www.erlang.org/doc) such as `kernel` and `stdlib`.

## 11.2. Working with dependencies

Depending on third-party libraries is an important feature. As soon as you start developing more complicated projects, you'll probably want to use various libraries, such as web frameworks, the JSON parser, and database drivers, to name a few examples.

As mentioned in the previous section, dependencies have two types: build-time and runtime. With the former, you tell `mix` to fetch external code and compile it while building your project. The latter is an OTP construct where you specify which applications must be running in order for your application to work.

Let's see this in action. Chapter 9 introduced a process registry that allows you to discover processes via rich aliases—something that isn't possible with plain, locally registered aliases that are atoms. I've mentioned that you don't need to implement this manually. A popular third-party library called gproc (https://github.com/uwiger/gproc) does exactly this (and much more). What you'll do is replace your own process registry with gproc.

### 11.2.1. Specifying dependencies

To do this, you must first visit mix.exs and specify your build-time dependency, as illustrated in the next listing.

**Listing 11.3. Reusing gproc in your project (gproc_todo_app/mix.exs)**

```
defmodule Todo.Mixfile do
  ...
  defp deps do
    [
      {:gproc, "0.3.1"},
      {:meck, "0.8.2", only: :test}
    ]
  end
end
```

Specifies a build-time dependency

Here, you instruct `mix` to fetch the dependency. Dependencies are fetched from Elixir's external package manager, which is called Hex (http://hex.pm). Other possible dependency sources include the GitHub repository, a git repository, or a local folder.

At this point, you've specified that your project depends on an external library, so you need to fetch your dependencies. This can be done by running `mix deps.get` from the command line. For this to work, you'll need a Hex client. If it isn't available, `mix` will offer to install this automatically for you.

Running `deps.get` fetches all dependencies (recursively) and stores the reference to the exact version of each dependency in the mix.lock file, unless mix.lock already exists on the disk, in which case this file is consulted to fetch the proper versions of dependencies. This ensures reproducible builds across different machines, so make sure you include mix.lock into the source control where your project resides.

Now that you've fetched all of your dependencies, you can build the entire system by running `mix compile`. Gproc is an Erlang library that isn't itself a `mix` project; instead, it relies on the `rebar` tool (a sort of `mix` counterpart in Erlang). But `mix` detects this situation and either uses your system's `rebar` (if you have one) or offers to install one.

In any case, it's good to know that `mix` can also work with Erlang libraries, which in turn means you have immediate access to Erlang's entire ecosystem. It's also worth mentioning that `mix` can detect a `Makefile` in your `deps` and run `make` to build such projects.

Because gproc is itself an OTP application that is used by your application at runtime, you need to specify the runtime dependency. This is done as follows.

**Listing 11.4. Depending on the `gproc` application at runtime (gproc_todo_app/mix.exs)**

```
defmodule Todo.Mixfile do
  ...

  def application do
    [
      applications: [:gproc],
      mod: {Todo.Application, []}
    ]
  end

  ...
end
```

Specifies a runtime dependency to gproc

This ensures that gproc is automatically started when you start your application.

### 11.2.2. Adapting the registry

With these preparations in place, you can start replacing the registry. Back in section 9.1.2, you created a registry that relies on via tuples. This allowed you to integrate nicely with `GenServer` and use aliases such as `{:via, Todo.ProcessRegistry, {:database_worker, 1}}` when creating processes or issuing requests to them.

Owing to this, and the fact that gproc also supports via tuples, it's simple to make the switch. Instead of using `{:via, Todo.ProcessRegistry, key}`, you can use `{:via, :gproc, key}` in the appropriate places (`Todo.DatabaseWorker` and `Todo.Server`).

There is a small twist due to the gproc interface. Gproc requires that keys are triplets in the form `{type, scope, key}`. You can check for details in the gproc documentation (http://mng.bz/sBck); but in this case, you need a triplet in the form of `{:n, :l, key}`, where `:n` indicates a unique registration (only one process can register itself under a given key) and `:l` stands for a local (single-node) registration. Of course, you can use an arbitrary term as a `key`.

Given all this, the changes are simple and not presented here. You can find them in gproc_todo_app/lib/todo/database_worker.ex and

gproc_todo_app/lib/todo/server.ex, where I adapted `via` tuples to the format just discussed.

With these modifications finished, you can remove the `Todo.ProcessRegistry` module. This in turn means you don't have to supervise the registry process (it resides in the gproc supervision tree) and can simplify your supervision tree. The top-level supervisor can now start the to-do system directly, as shown in the following listing.

**Listing 11.5. Simplified top-level supervisor (gproc_todo_app/lib/todo/supervisor.ex)**

```
defmodule Todo.Supervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, nil)
  end

  def init(_) do
    processes = [
      supervisor(Todo.Database, ["./persist/"]),
      supervisor(Todo.ServerSupervisor, []),
      worker(Todo.Cache, [])
    ]
    supervise(processes, strategy: :one_for_one)
  end
end
```

And that's it! You've successfully replaced your manual implementation of a process registry with a full-blown, battle-tested Erlang library. Let's quickly verify that it works:

```
iex(1)> bobs_list = Todo.Cache.server_process("Bob's list")
Starting to-do server for Bob's list
#PID<0.86.0>

iex(2)> Todo.Cache.server_process("Bob's list")
#PID<0.86.0>

iex(3)> Todo.Server.add_entry(
          bobs_list,
          %{date: {2013, 12, 19}, title: "Dentist"}
        )

iex(4)> Todo.Server.entries(bobs_list, {2013, 12, 19})
[%{date: {2013, 12, 19}, id: 1, title: "Dentist"}]
```

Repeated retrieval fetches the same process

In essence, gproc works roughly the same as your own process registry. It relies on ETS tables and sets up monitors to your processes to detect crashes and deregister them.

## 11.3. Building a web server

The time has finally come to introduce an HTTP interface into the to-do system. You'll implement a rudimentary server, wrapping only `entries` and `add_entry` requests.

The focus won't be so much on the details and finesse of web servers. Instead, my aim is to demonstrate how to work with OTP applications and present how everything connects in a simplistic simulation of a real-world system.

### 11.3.1. Choosing dependencies

You could of course implement the entire server from scratch, but that would be too much work. Instead, you'll reach for a couple of existing libraries to make your life easier. Note that it's not the purpose of this section to provide detailed descriptions of those libraries; they're just a simple means to an end (a basic working HTTP server). You're encouraged to research those libraries in more detail on your own.

Various web server frameworks and libraries are available for both Elixir and Erlang. The Erlang ones are definitely more mature; the Elixir libraries are currently (at the time of this writing) in a more unstable, unfinished state. But by the time this book reaches you, this may change, so you should definitely look at the state of Elixir web frameworks. Currently, the most active and advanced Elixir web framework is Phoenix (https://github.com/phoenixframework/phoenix).

In this example, you'll use two HTTP libraries. The first is an Erlang library called Cowboy (https://github.com/extend/cowboy); it's a fairly lightweight and efficient HTTP server library that is a popular and frequent choice for HTTP servers in Erlang ecosystem.

But you won't use Cowboy directly. Instead, you'll interface it via the Plug library (https://github.com/elixir-lang/plug)—a project maintained by Elixir core team. Plug is somewhat similar to Ruby's Rack or Clojure's Ring. The aim of Plug is to provide a unified API that

abstracts away the web library of a framework. Moreover, Plug introduces the concept of stackable *plugs*—middleware modules or functions that can be injected into the request-handling pipe and that intervene in various phases of request processing.

Plug doesn't implement the HTTP server library itself. Instead, Plug requires an adapter to an existing library. Out of the box, an adapter to the Cowboy library is provided, and this is what you'll use here.

Finally, you'll use another test-only dependency called `httpoison`. This is an HTTP client library that is internally used to test the HTTP server. As mentioned, I won't deal with tests in this book, but you may want to look at the todo_web/test/http_server_ test.exs file to see a simple example of an (admittedly hacky) HTTP server test.

Let's introduce these dependencies to the mix.exs file. The relevant changes are given next.

**Listing 11.6. External dependencies for the web server (todo_web/mix.exs)**

```
defmodule Todo.Mixfile do
  def application do
    [
      applications: [:gproc, :cowboy, :plug],     <-- Runtime dependencies
      mod: {Todo.Application, []}
    ]
  end

  defp deps do
    [
      {:gproc, "0.3.1"},
      {:cowboy, "1.0.0"},                          | Compile-time
      {:plug, "0.10.0"},                           | dependencies
      {:meck, "0.8.2", only: :test},
      {:httpoison, "0.4.3", only: :test}
    ]
  end
end
```

### 11.3.2. Starting the server

With the dependencies configured, you can now run `mix deps.get` and start implementing the HTTP interface. As mentioned, your

primary interface to work with Cowboy is Plug. Plug is a reasonably complex library, and I won't provide an in-depth treatment here. The focus is on getting a basic version running and understanding how all the pieces work together.

To implement a Plug-powered server, you can introduce a dedicated module. The basic skeleton is provided in the following listing.

**Listing 11.7. Starting the server (todo_web/lib/todo/web.ex)**

```
defmodule Todo.Web do
  ...

  def start_server do
    Plug.Adapters.Cowboy.http(__MODULE__, nil, port: 5454)
  end

  ...
end
```

Starts the HTTP server

In `start_server/0`, you start the Cowboy-powered HTTP server on a desired port. The second argument is an arbitrary term that will be passed to your handler functions on each HTTP request.

It's important to know that the Cowboy server will run various processes. There will be at least one process that listens on a given port and accepts requests. Then, each distinct TCP connection will be handled in a separate process, and your callbacks (which you have to implement) will be invoked in those request specific processes.

Notice that, despite this elaborate process structure, you don't set up a supervision tree. You call the `http/3` function and disregard the result. Why is that? Because Cowboy takes it upon itself to supervise processes it creates. To be more accurate, most processes reside in the supervision tree of the Ranch application, an internal dependency of Cowboy where TCP/SSL communication is handled.

**Tip**

You can observe the state of your system via a graphical tool called `observer` that ships with Erlang/OTP. If your Erlang distribution is properly built, you can run `:observer.start` from the `iex` shell. In the observer GUI, you'll be able to inspect running application and their supervision trees.

This begs one question, though. Cowboy takes care of supervising its own processes, so where do you invoke `Todo.Web.start_server/0`? The simplest approach is to call it manually when you start the application. The idea is illustrated next.

**Listing 11.8. Starting the HTTP server (todo_web/lib/todo/application.ex)**

```
defmodule Todo.Application do
  use Application

  def start(_, _) do
    response = Todo.Supervisor.start_link
    Todo.Web.start_server                    <-- Starts the HTTP server
    response
  end
end
```

Notice the explicit ordering. First you start the internal system (the to-do process structure), and only then do you start the HTTP server. This makes sense, because the server relies on the supporting processes, so you need to start those processes first.

One final point. Remember that applications are singletons—you can start only one instance of a distinct application in a running BEAM instance. This fact doesn't mean you can run only one HTTP server in your system. The Cowboy application can be considered a factory for HTTP servers. When you start the Cowboy application, the supervision tree is set up, but no HTTP server is started yet.

Only when you call `Plug.Adapters.Cowboy.http/3` is a separate set of processes started and placed in the Ranch supervision tree. This means you can call `http/3` as many time as you want (providing a different port, of course). You could, for example, just as easily start an additional HTTPS server or run multiple unrelated servers from the same BEAM instance.

### 11.3.3. Handling requests

With the scheme set-up, you can start handling some requests. Let's introduce support for `add_entry`. This will be a POST request, but for the sake of simplicity, you'll transfer all parameters via URL. The example request then looks like this:

```
http://localhost:5454/add_entry?list=bob&date=20131219&title=Dentist
```

Let's begin by setting up a route for this request. The skeleton is provided next.

**Listing 11.9. Setting up a route for `add_entry` (todo_web/lib/todo/web.ex)**

```
defmodule Todo.Web do
  use Plug.Router                    <-- Includes boilerplate

  plug :match
  plug :dispatch                     | Sets up plugs

  ...

  post "/add_entry" do               <-- Specifies a handler
    ...
  end

  ...
end
```

There are some strange constructs here, which are part of how Plug is used. You don't need to understand all this, because it isn't the aim of this exercise, but let's look at a simple explanation.

Calling `use Plug.Router` adds some functions to your module. This is similar to how you used the Elixir behaviour helper (by calling `use GenServer`). For the most part, the functions imported will be used internally by Plug.

Constructs such as `plug :match` and `plug :dispatch` deserve special mention. These calls will perform some additional compile-time work that will allow you to match different HTTP requests on your server. These constructs are examples of Elixir macro invocations, which will be resolved at compilation time. As a result, you'll get some additional functions in your module (which are again used only by Plug).

Finally, you call a `post` macro, which suspiciously resembles a

function definition. In fact, under the hood, this macro will generate a function that is used by all the other generated boilerplate you got by calling the `plug` macro and `use Plug.Router`. To make this clearer, the generated function will look roughly like the following:

```
defp do_match("POST", ["add_entry"]) do
  fn(conn) ->
    ...                          <-------- Code passed to the post macro
  end
end
```

This `do_match` function is invoked by the `match` function, which you have in your module courtesy of `plug :match`.

Of course, if you issue multiple calls to the `post` macro, you'll have multiple clauses for this `do_match` function, each clause corresponding to the single route you're handling. You'll end up with something like this:

```
defp do_match("POST", ["add_entry"]), do: ...
defp do_match("POST", ["delete_entry"]), do: ...
...
defp do_match(_, _), do: ...          <-------- Handles all other requests
```

Ultimately, the code you provide to `post "/add_entry"` is invoked when an HTTP POST request with an `/add_entry` path arrives at your server.

Let's see the implementation of the request handler, provided in the following listing.

**Listing 11.10. Implementing the `add_entry` request (todo_web/lib/todo/web.ex)**

```
defmodule Todo.Web do
  ...

  post "/add_entry" do
    conn
    |> Plug.Conn.fetch_params
    |> add_entry
    |> respond
  end

  ...
end
```

Notice that in the request handler you use the `conn` variable, which doesn't exist anywhere. This variable is brought to you by the `post` macro, which generates this variable and binds it to the proper value.

As the name implies, the `conn` variable holds your connection. This is an instance of a `Plug.Conn` structure that holds your TCP socket, together with various information about the state of the request you're processing. From your handler code, you must return the modified connection that will hold response information such as its status and body.

Notice how you thread the variable through various functions, some of which are your own (`add_entry` and `respond`). Almost all operations you can perform on connection will return the modified connection structure. Somewhat surprisingly, this also holds for some getter functions. For example, `Plug.Conn.fetch_params` returns a new version of the connection structure with the field `params` containing request parameters (in the form of a map). Relying on `[]` map-access syntax, you can call `conn.params[param_name]` to fetch individual parameters. This is essentially a caching technique. Plug caches the result of `fetch_params` in the connection struct, so repeated calls to `fetch_params` won't result in excessive parsing.

You'll also adapt to this style, by making your custom functions (`add_entry/1` and `respond/1`) accept the connection and return the modified one. Let's see the implementation of `add_entry/1`.

**Listing 11.11. Implementation of `add_entry/1` (todo_web/lib/todo/web.ex)**

```
defmodule Todo.Web do
  ...
  defp add_entry(conn) do
    conn.params["list"]
    |> Todo.Cache.server_process
    |> Todo.Server.add_entry(
         %{
           date: parse_date(conn.params["date"]),
           title: conn.params["title"]
         }
       )

    Plug.Conn.assign(conn, :response, "OK")
  end

  ...
end
```

Calls into the existing to-do system

Calls into the existing to-do system

Puts the response into the connection

Here you take parameters from the connection and reuse existing to-do services that you already implemented in previous chapters. Notice the call to `parse_date/1`. This is a custom function that transforms a string date (such as `"20131219"` into an internal tuple you're using `{2013, 12, 19}`).

The most important part is the final call to `Plug.Conn.assign/3`. This technique allows you to place custom key-value information in the connection structure. Remember, you have to return the modified version of the plug connection from your function. Therefore, you can't return another value, but you can use `Plug.Conn.assign/3` to place arbitrary data in the connection structure. In this case, you're setting the custom `:response` field to a value that you'll use later while making the final response.

The final thing to do is implement the `respond/1` function. The code is given in the next listing.

**Listing 11.12. Implementing `respond/1` (todo_web/lib/todo/web.ex)**

```
defmodule Todo.Web do

  ...

  defp respond(conn) do
    conn
    |> Plug.Conn.put_resp_content_type("text/plain")
    |> Plug.Conn.send_resp(200, conn.assigns[:response])
  end

  ...
end
```

This is mostly straightforward. You set the response type to plain text and send the response. The only thing worth pointing out is the call to `conn.assigns[:response]`, where you read a value that you set earlier in `add_entry/1`, when you called `Plug.Conn.assign(conn, :response, "OK")`.

The same approach is used to support the `entries` request (except that it's handled as a GET request, so it uses the `get` macro instead of `post`). For the sake of brevity, I'll omit the code, but you can find it in the same file (todo_web/lib/todo/web.ex).

In any case, you can now test the system. You can start the system with `iex -S mix` and issue requests. For example, this is what you get using a command-line `curl` tool:

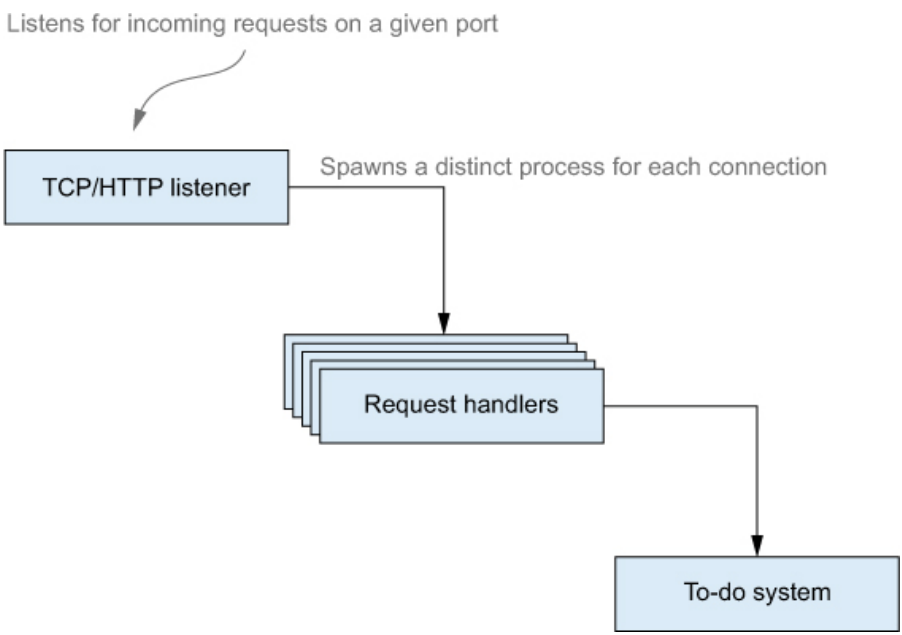
```
$ curl -d "" \
  "http://localhost:5454/add_entry?
list=bob&date=20131219&title=Dentist"
"OK"
$ curl "http://localhost:5454/entries?list=bob&date=20131219"
2013-12-19    Dentist
```

This proves that your system is working, but let's see how everything combines.

### 11.3.4. Reasoning about the system

First, let's look at how the HTTP server works. The simplified idea is illustrated in figure 11.1.

**Figure 11.1. Requests are handled in separate processes**


Listens for incoming requests on a given port
TCP/HTTP listener
Spawns a distinct process for each connection
Request handlers
To-do system

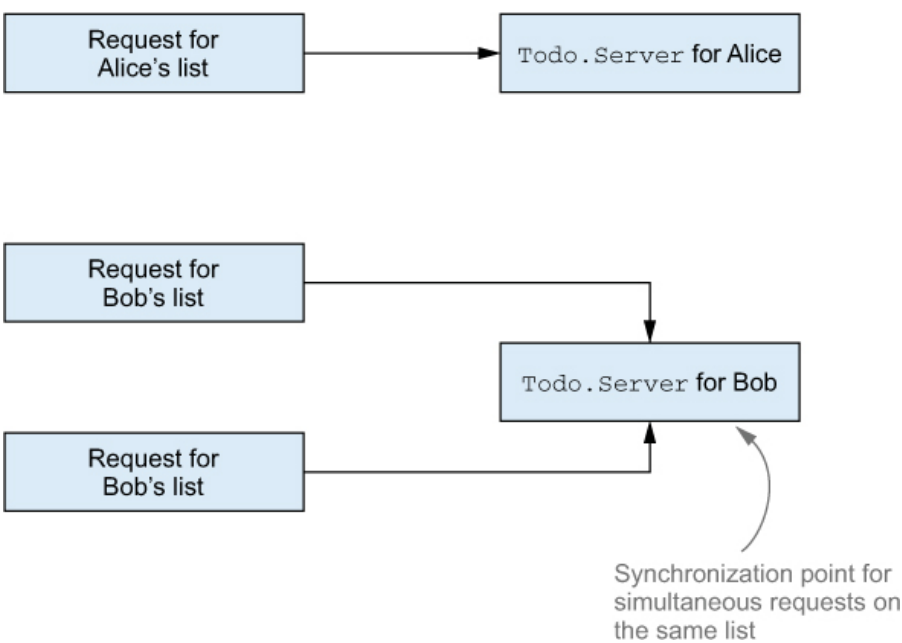
The most important thing to notice is that each connection is managed in a distinct process. In practice, this means different requests are handled in different processes. There is no special magic here—this is

how the underlying Cowboy web server is implemented. It uses one process to listen on a port, and then it spawns a separate process for each incoming request.

This architecture has all sort of benefits due to the way BEAM treats processes. Because processes are concurrent, CPU resources are maximally used, and the system is scalable. Because processes are lightweight, you can easily manage a large number of simultaneous connections. Moreover, thanks to the BEAM scheduler being preemptive, you can be certain that occasional long-running, CPU-intensive requests won't paralyze the entire system. Finally, due to process isolation, a crash in a single request won't affect the rest of the system.

Processes also make it easy to reason about the system. For example, you can be certain that independent requests won't block each other, whereas multiple requests on the same to-do list are synchronized, as illustrated in figure 11.2.

**Figure 11.2. Independent simultaneous requests are handled concurrently, whereas requests on the same to-do list are synchronized.**


Request for Alice's list
Todo.Server for Alice
Request for Bob's list
Todo.Server for Bob
Request for Bob's list
Synchronization point for simultaneous requests on the same list

This is due to the way the to-do cache works. Whenever you want to manipulate Bob's list, you first ask the to-do cache to return a pid of the process in charge. All requests for Bob's list go through that same process and are therefore handled one by one.

#### Performance

It's worth spending some time discussing the system's performance. In its current implementation, the system isn't particularly efficient.

A 30-second load test with 100 concurrent connections over 1,000 to-do lists gives a throughput of about 10,000 requests per second. This is a decent result, given that you're testing database operations and dynamic response generation. But looking at CPU usage, it takes some time before all cores get to 100% of usage. Moreover, a series of timeouts takes place while calling `Todo.Cache.server_process/1`. Finally, with a longer test, memory consumption significantly increases and forces the system to go into swap.

This isn't completely unexpected, because you made some suboptimal ad hoc decisions along the way. There are three primary causes for this poor performance:

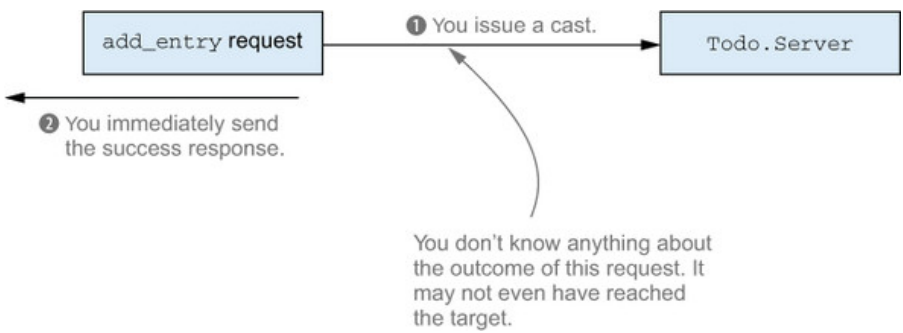
- You use an overly simplistic, strictly disk-based database.
- On every list modification, you store the entire list.
- The to-do list abstraction isn't efficient when doing date-based lookups. You do sequential searches over all list elements, which means the complexity of a date-based lookup is O(n).

Fixing those issues isn't particularly difficult, but it requires some changes, which I don't present here. I did a simplistic optimization, and the solution is available in the todo_web_optimized folder, if you want to look at it. The most important changes are in database.ex and database_worker.ex (an Mnesia database is used instead of a plain file), list.ex (internal reorganization of a to-do list abstraction), and server.ex (adapting to the new to-do list and reducing the amount of persisted data on each change). The consequence of those changes is that a 2-minute load test maintains a consistent throughput of 20,000 requests per second, and fully utilizes all CPU cores.

Let's discuss some effects that calls and casts may have on your system. To refresh your memory, *casts* are fire-and-forget kinds of requests. A caller sends a message to the server and then immediately moves on to do something else. In contrast, a *call* is a blocking request, where a caller waits for the server to respond.

Remember that you've opted to use casts for all operations except where you need to return a response. This was a somewhat arbitrary decision, made mostly for didactic purposes. In reality, casts have a drawback: you don't know what happened with your request. This in turn means you may be giving false responses to end users, as illustrated in figure 11.3.

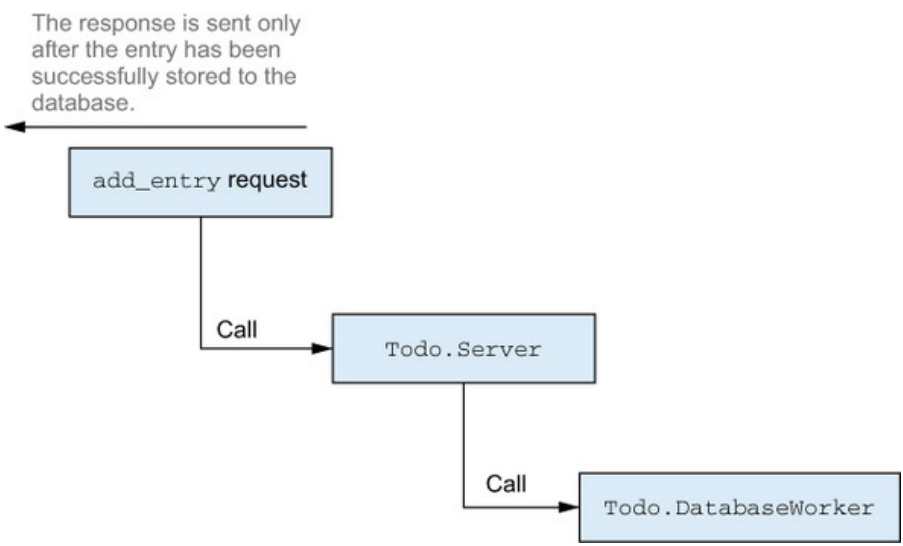
**Figure 11.3. Using casts reduces the certainty of your responses.**


add_entry request
1 You issue a cast.
Todo.Server
2 You immediately send the success response.
You don't know anything about the outcome of this request. It may not even have reached the target.

Because you use a cast to add a to-do entry in your system, you have no way of knowing what happened with your request. So when you're telling the end user that you succeeded, this is a guess rather than a truthful statement.

Obviously, the simple way to resolve this is to use calls, which are synchronous, meaning the client waits until the response arrives, as illustrated in figure 11.4.

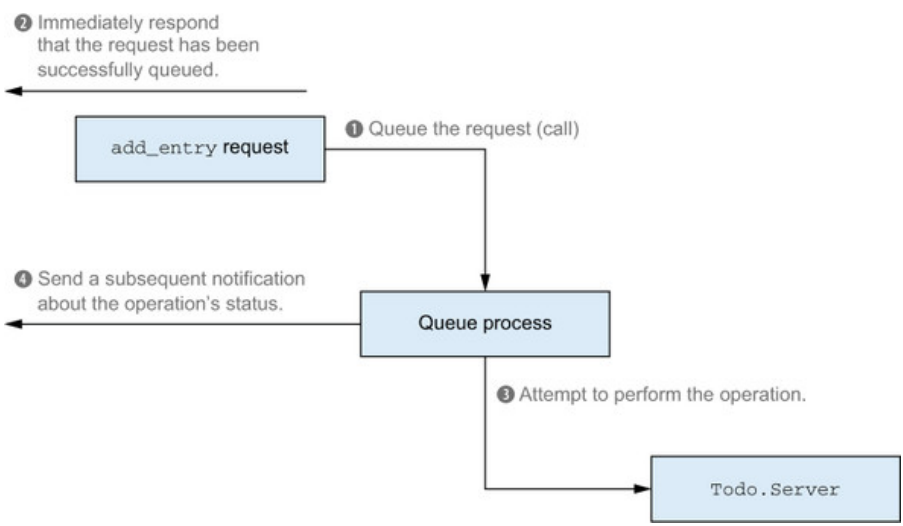
**Figure 11.4. Using calls promotes consistency but reduces the responsiveness of the system.**




This approach is more consistent: you return success only when you're certain that the entry has been stored. But the downside is that the entire system now depends on the throughput of database workers—and as you may recall, you're running only three workers, and you're using a pretty inefficient database.

This can be resolved by introducing an intermediate process. The idea is to provide an immediate response stating that the request has been queued. Then you do your best to process the request, and you send a subsequent notification about the request's status. See figure 11.5.

**Figure 11.5. Queuing the request and sending a notification about the status**


❷ Immediately respond that the request has been successfully queued.
add_entry request
❶ Queue the request (call)
❹ Send a subsequent notification about the operation's status.
Queue process
❸ Attempt to perform the operation.
Todo.Server

This scheme is definitely more elaborate and involved, so it's not appropriate for simpler cases where using plain calls is enough. But in cases when the load is very high and end-to-end operations can take longer, introducing an intermediate process may be beneficial. This process can increase the responsiveness of the system while retaining consistency. Moreover, this process can serve as a proactive agent in dealing with congestion and increased load. If the system becomes overloaded and the queue starts piling up, you can refuse to take more requests in the queue process until you regain some breathing space.

As always, no single approach works for all cases. Using calls can be a reasonable first attempt because it promotes consistency. Later, you can easily switch to casts or introduce an intermediate process, depending on the specific situation.

At this point, you're finished implementing a basic HTTP server. One small problem remains, the listening port is currently hard-coded in the web.ex file. Let's look at how you can make this configurable.

## 11.4. Managing the application environment

It's possible to manage application parameters via a construct called an *application environment*—a key-value, in-memory store where both keys and values are arbitrary, complex Elixir terms. The nice thing about environments is that they let users of your application provide alternative values before the application is started. This plays an important role when assembling deployable releases, because it allows administrators to tweak the behavior of the system without having to recompile the code.

**Note**

You should always be aware of the distinction between the `mix` (project) environment and the application environment. The former is a simple identifier that can be used to tweak the outcome of compilation. In contrast, an application environment is an OTP construct that allows you to make your application configurable without the need to recompile the code.

Let's look at a simple example. The `iex` shell you've been using throughout this book is itself an OTP application. Various parameters of the `iex` runtime can be configured via its application environment. For example, you can use `Application.put_env/3` to modify a parameter such as the shell default prompt:

```
iex(1)> Application.put_env(:iex, :default_prompt, "Elixir>")
:ok

Elixir>
```

Sets the application environment at runtime

When you call `Application.put_env/3`, you provide the application atom (`:iex`) followed by the key and the value of the setting you want to change. The effect is immediately visible in the prompt.

It's also possible to provide an application environment key while starting the system. You just need to include the `--erl '-an_app_name key value'` parameter when starting the system. Because this is an Erlang option, both the key and value must be provided using Erlang syntax rather than Elixir.

Therefore, to change the `iex` prompt, you can issue the following command:

```
$ iex --erl '-iex default_prompt <<"Elixir>">>'
Elixir>
```

Adds a key-value pair to the iex application environment

The whole point of application environments is to allow clients (for example, another OTP app or an administrator running the system) to change parameters of your application. For example, the `todo` web system currently listens on a hard-coded port 5454. If you want to alter this, you must change and recompile the code and then deploy the new version.

Alternatively, you can require that the port be specified via an application environment. The change is simple, as shown next.

**Listing 11.13. Retrieving a port from application (todo_env/lib/todo/web.ex)**

```
defmodule Todo.Web do
  ...
  def start_server do
    case Application.get_env(:todo, :port) do
      nil -> raise("Todo port not specified!")
      port ->
        Plug.Adapters.Cowboy.http(__MODULE__, nil, port: port)
    end
  end

  ...
end
```

Retrieves the port from the todo application environment

The code is straightforward. You read the port from the environment using `Application .get_env/2`, and you raise an error if the port setting doesn't exist.

How do you provide the port setting? A simple way is to use the previously mentioned command-line syntax `--erl -app key value`:

```
$ iex --erl "-todo port 5454" -S mix
```

Another option is to list the defaults in the application configuration of the mix.exs file:

```
defmodule Todo.Mixfile do
  ...
  def application do
    [
      applicatio [:gproc, :cowboy, :plug],
      mod: {Todo.Application, []},
      env: [
        port: 5454
      ]
    ]
  end

  ...
end
```

Provides a default application environment from mix.exs

This has the nice benefit of providing all the defaults in a single place, which makes it possible to quickly find all possible configurable parameters. These defaults are assembled into `todo.app` and used at runtime by OTP to set the default application environment. Clients can still override those settings—for example, using the previous

command-line syntax or by changing it dynamically by calling `Application .put_env/3`.

But if your system is more complex and exposes a lot of configurable parameters, mix.exs can quickly become large and difficult to analyze. In practice, you should keep this file simple, because it acts as a sort of manifest of your project, describing the application with its dependencies and version requirements. Another configuration mechanism is available in the config/config.exs file, as shown in the following listing.

**Listing 11.14. Using config.exs (todo_env/config/config.exs)**

```
use Mix.Config                  <-- Brings in compile-time helpers
config :todo, port: 5454        <-- Adds an application's environment value
```

A nice benefit of config.exs is that you can also specify settings for other applications you depend on, by listing `config :some_app, ...` statements.

## 11.5. Summary

This concludes our tour of OTP applications. We've covered a lot of ground, so let's recap the most important points:

- An OTP application is a reusable component. The application can run the entire supervision tree or just provide utility modules (as a library application).
- In a single BEAM instance, an application is a singleton—it either runs or doesn't run.
- A non-library application is a callback module that must start the supervision tree.
- Applications allow you to specify runtime dependencies to other applications. This isn't the same as compile-time dependencies that allow you to fetch external code and compile it from your project.
- Application environments let clients tweak parameters of your applications without needing to change and recompile the application code.

At this point, you have implemented a tangible system: an HTTP server that can perform basic manipulations of multiple to-do lists. Next, we'll look at distributed systems, and you'll make the to-do system scale across multiple machines.

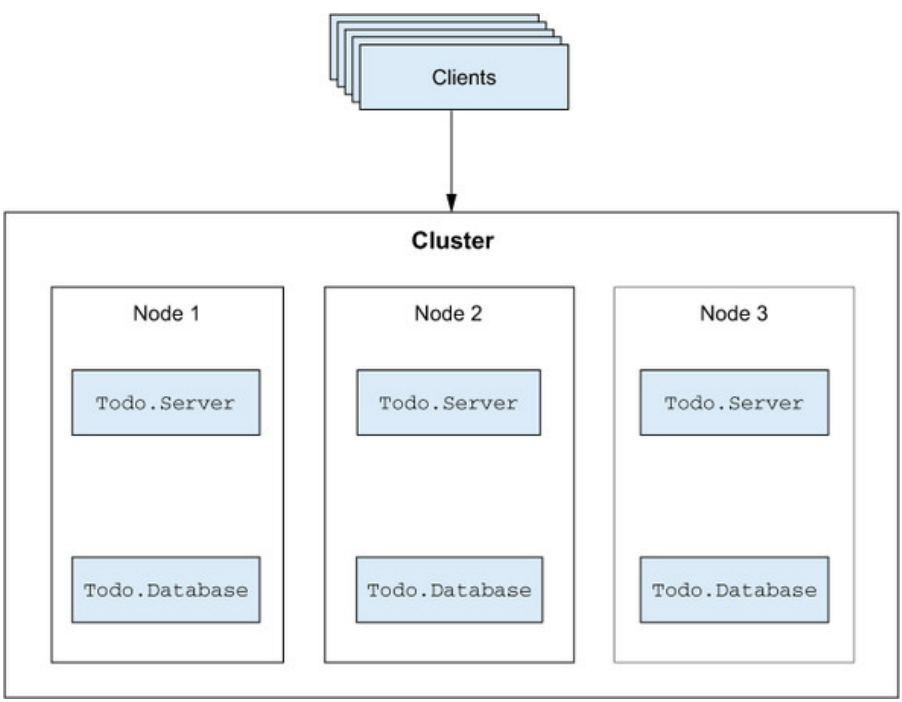
# Chapter 12. Building a distributed system

*This chapter covers*

- Working with distribution primitives
- Building a fault-tolerant cluster
- Network considerations

Now that you have a to-do HTTP server in place, it's time to make it more reliable. To have a truly reliable system, you need to run it on multiple machines. A single machine represents a single point of failure, because a machine crash leads to a system crash. In contrast, in a cluster of multiple machines, a system can continue providing service even when individual machines are taken down. Moreover, by clustering multiple machines, you have a chance of scaling horizontally. When demand for the system increases, you can add more machines to the cluster to accommodate the extra load. This idea is illustrated in figure 12.1.

**Figure 12.1. The to-do system as a cluster**



Here you have multiple nodes sharing the load. If a node crashes, the remaining load will be spread across survivors, and you can continue to provide service. If the load increases, you can add more nodes to the cluster to take the extra load. Clients access a well-defined endpoint and are unaware of internal cluster details.

Distributed systems obviously offer significant benefits, and Elixir/Erlang gives you some simple and yet powerful distribution primitives. The central tools for distributed Erlang-based systems are processes and messages. You can send a message to another process regardless of whether it's running in the same BEAM instance or on another instance on a remote machine.

Don't confuse this with a traditional RPC approach, where a remote call is wrapped to look like a local call. Erlang and, by extension, Elixir take the opposite route, and their distributed nature appears early in the game. If you think about it, a typical concurrent system that runs a multitude of processes can already be considered distributed.

Much like remote components, processes live their own lives and run in total isolation from each other. Issuing a request to another local process can be considered a remote call, and message passing has much in common with remote network communication. In the basic version, you send a message and don't know anything about its outcome. You can't even be sure whether the message will reach the target. If you want stronger guarantees, you can design the protocol to make the target send you a response (for example, by using a synchronous call). Moreover, you must take into account that there is a cost to passing a message (the contents are copied), and this property sometimes affects the design of the communication protocol among multiple processes.

All these properties are common to the Erlang concurrency model and distributed systems, and you need to take them into consideration. The good news is then that a properly designed concurrent system is in many ways ready to be distributed across multiple machines. Of course, this transformation is by no means free. Distributed systems introduce an additional set of non-trivial challenges that need to be tackled. But thanks to the simple distribution building blocks that are available, many of which you're already familiar with, you can focus

on the core challenges of distributed systems.

As you'll see in this chapter, it doesn't take much to turn your to-do system into a basic fault-tolerant cluster. In order to do this, you need to become familiar with basic distribution primitives.

## 12.1. Distribution primitives

BEAM-powered distributed systems are built by connecting multiple *nodes* into a cluster. A node is a BEAM instance that has a name associated with it. You can start multiple nodes on the same host machine or on different machines, and you can connect those nodes. Once the nodes are connected, you can communicate between different processes on different nodes by relying on the familiar message-passing mechanism.

### 12.1.1. Starting a cluster

To set up a cluster, you need to start a couple of nodes. Starting a node can be as simple as using the `--sname` parameter while starting the shell:

```
$ iex --sname node1@localhost          <---- Provides the node name
iex(node1@localhost)1>                 <---- The shell reports the node name.
```

Using `--sname` turns your BEAM instance into a node with the name `node1@localhost`. The part before the `@` character is a prefix that uniquely identifies a node on a single machine. The second part (`localhost`) identifies the host machine. If you omit the host part, the host machine's name is automatically used.

The `--sname` parameter is used to set a *short name* in which the host machine is identified only by its name. It's also possible to provide a *long name* in which the host machine is identified by a fully qualified symbolic name or an IP address. This will be discussed in more detail in the final section of this chapter.

Once you've started a node, you can obtain its name by calling the `Kernel.node/0` function:

```
iex(node1@localhost)1> node
:node1@localhost                    <-- The name of this node
```

As you can see from the output, a node name is represented internally as an atom.

Using a node usually makes sense when you want to connect it to another node. Let's try this. Keep `node1` running, and start another OS shell session. Now, start `node2` and connect it to `node1`:

```
$ iex --sname node2@localhost

iex(node2@localhost)1> Node.connect(:node1@localhost)   <-- Connecting to another node
true
```

The argument to `Node.connect/1` is an atom that represents the target node name. When `Node.connect/1` is invoked, BEAM tries to establish a TCP connection with the target BEAM instance. Once the connection is established, nodes are considered to be connected, and all communication between them takes place via this connection.

You can prove that nodes are connected by calling `Node.list/0`, which returns a list of all nodes connected to the current one (the current node isn't listed). Trying this on `node1` and `node2` gives the expected results:

```
iex(node1@localhost)2> Node.list
[:node2@localhost]                  <-- Nodes connected to node1

iex(node2@localhost)2> Node.list
[:node1@localhost]                  <-- Nodes connected to node2
```

It's of course possible to connect multiple nodes. In fact, BEAM by default tries to establish a fully connected cluster. If you start a node `node3` and connect it to `node2`, a connection is established to all other nodes that `node2` is connected to:

```
$ iex --sname node3@localhost

iex(node3@localhost)1> Node.connect(:node2@localhost)

iex(node3@localhost)2> Node.list
[:node2@localhost, :node1@localhost]   <-- node3 is connected to all nodes.
```

This is useful in scenarios where you want to set up a fully connected cluster of multiple nodes. Adding a new node to such a cluster amounts to establishing a connection to a single node from the cluster. The new node will then automatically connect to all nodes in the cluster.

To get the list of all nodes in a cluster, including the current one, you can use `Node.list/1`:

```
iex(node1@localhost)3> Node.list([:this, :visible])
[:node1@localhost, :node2@localhost, :node3@localhost]
```

The `:this` option states that you want the current node to appear in the list. The `:visible` option indicates that you also want to get the list of all *visible* nodes. It's possible to start a node as *hidden*, as I'll explain in the last section of this chapter.

**Detecting disconnected nodes**

Node disconnection deserves a special mention. After the connection is established, each node periodically sends *tick messages* to all of its connected peers, to check whether they're still alive. All nodes that fail to respond to four consecutive tick messages are considered to be disconnected and are removed from the list of connected nodes. There is no automatic attempt to reconnect those nodes. But it's possible to register and receive notifications when a node is disconnected, using the `Node.monitor/1` function (http://mng.bz/ge5f). Moreover, you can monitor all node connections and disconnections with the help of `:net_kernel.monitor_nodes/1,2` (http://mng.bz/9799). I'll demonstrate how this works a bit later when I discuss network partitions.

### 12.1.2. Communicating between nodes

Once you have some nodes started and connected, you can make them cooperate. A simple way to try this is to use `Node.spawn/2`, which receives a node name (an atom) and a lambda. The function then spawns a new process on the target node and runs the lambda in that process.

For example, from `node1`, you can spawn a process on `node2`:

```
iex(node1@localhost)4> Node.spawn(
                         :node2@localhost,
                         fn -> IO.puts "Hello from #{node}" end
                       )
Hello from node2@localhost
```

Target node

Runs on the target node

The output proves that lambda has been executed on another node.

**Group leader process**

Something unexpected is happening in this example. Even though the lambda has been executed on `node2`, the output is printed in the shell of `node1`. How is this possible? The reason lies in how Erlang does standard I/O operations.

All standard I/O calls (such as `IO.puts/1`) are forwarded to the *group leader*—a process that's in charge of performing the actual input or output. A spawned process inherits the group leader from the process that spawned it. This is true even when you're spawning a process on another node. Therefore, your process may run on `node2`, but its group leader is still on `node1`. As a consequence, the string to be printed is created on `node2` (as the string contents prove), but the output is printed on `node1`.

Another important primitive is the ability to send messages to processes regardless of their location. This property is also known as *location transparency*. The send operation works exactly the same regardless of the node on which the target process is running.

Let's look at a simple example. From `node1`, you'll start a computation that runs on `node2` and send the result back to `node1`:

```
iex(node1@localhost)5> caller = self
iex(node1@localhost)6> Node.spawn(
                         :node2@localhost,
                         fn -> send(caller, {:response, 1+2}) end
                       )
iex(node1@localhost)7> flush
{:response, 3}
```

Sends the response back to the caller

The response is received on the caller.

This example clearly resembles standard usage of processes. You spawn a process on a remote node, and then, from this spawned process, you send the message back to the caller. Notice how the `caller` variable is used. Even though the lambda runs on another node, the closure mechanism still works.

Finally, you use the `iex` shell `flush` helper, which takes all messages from the current process mailbox and prints them to the console. This proves that the messages have been received on the caller node.

There are no limits to what can be sent as a message. Whatever works in the same BEAM instance will work across different instances (with a small caveat, described in the sidebar). When the destination process is on another node, the message is encoded using `:erlang.term_to_binary/1` and decoded on the target node with `:erlang.binary_to_term/1`.

**Avoid spawning lambdas or sending them to different nodes**

You can spawn lambdas from your shell, which is a somewhat special case because shell-defined lambdas embed their own code and are interpreted dynamically on each invocation. In contrast, lambdas that are defined in module functions are immediately compiled and can be spawned remotely (or sent to a remote node via a message) only if both nodes are powered by exactly the same compiled code. These requirements are hard to satisfy if you start running a multinode cluster and then need to update the code. You can't simultaneously upgrade all the nodes in the cluster, so at some point the code on both nodes will differ.

Therefore, it's generally better to avoid passing lambdas to a remote node. Instead, you should use the `Node.spawn/4` function, which accepts an MFA (module, function, arguments list) that identifies a function to be invoked on the target node. This is safe to use as long as the module exists on the target node and exports the corresponding function.

In a multinode environment, the term *local registration* finally starts to

make sense. When you register a process locally, the scope of registration is only the current node. This means you can use the same alias on different nodes (but only once per each node). For example, let's register shell processes of both `node1` and `node2`:

```
iex(node1@localhost)8> Process.register(self, :shell)
true

iex(node2@localhost)3> Process.register(self, :shell)
true
```

Calling `send(:shell, some_message)` will send the message to either `node1` or `node2`, depending on the node where you invoke `send`.

It's possible to reference a locally registered process on another node by using `{some_alias, some_node}`. For example, to send a message from `node1` to `node2` shell, you can do this:

```
iex(node1@localhost)9> send(
                         {:shell, :node2@localhost},
                         "Hello from node1!"
                       )
```

Identifies a process registered on another node

Then, on `node2`, you can verify that a message is received:

```
iex(node2@localhost)4> flush
"Hello from node1!"
```

You can also use the `{some_alias, some_node}` form while making `GenServer`-powered request (casts and calls). Finally, there are two special functions, `GenServer.abcast/3` and `GenServer.multi_call/4`, that let you issue a request to all locally registered processes on given nodes.

### 12.1.3. Process discovery

Process discovery is a very important operation in a cluster. But this same operation is used in cluster-less mode as well. In fact, distributed system or not, the typical pattern of process communication is always the same:

1. A client process must obtain the server's pid.
2. A client sends a message to the server.

In step 1, you discover a process. You used a form of discovery with the custom process registry that you implemented in chapter 9 and then later in chapter 11 replaced with the third-party `gproc` library.

Even in a single-node system, you must somehow find the target process pid. This doesn't change in a distributed setting. But you must use another means of discovery, because the process registry (and to some extent `gproc`) isn't cluster aware and works only in the scope of a local node.

#### Global registration

The simplest way to do cluster-wide discovery is to use the `:global` module (http://erlang.org/doc/man/global.html), which provides global alias registration facility. For example, if you run the to-do system as a multinode cluster, you may want to run exactly one process per single to-do list (unless you aim for redundancy, of course). Global name registration allows you to achieve this. As an example, you can register the `node1` shell process to act as the process responsible for handling Bob's to-do list:

```
iex(node1@localhost)10> :global.register_name({:todo_list, "bob"},
self)
:yes
```

The global (cluster-wide) alias of the current process is now `{:todo_list, "bob"}`. The result (`:yes`) means global registration is successful. At this point, all processes on all nodes in the cluster can find the process registered under this alias. Attempting to globally register the `node2` shell process under the same alias will fail:

```
iex(node2@localhost)7> :global.register_name({:todo_list, "bob"},
self)
:no
```

You can use `:global.whereis_name/1` to find the process:

```
iex(node2@localhost)8> :global.whereis_name({:todo_list, "bob"})
#PID<7954.59.0>
```

### How global registration works

There is no special magic to global registration. It's implemented in pure Erlang, and you can reimplement it yourself in Elixir. It's just an elaborate, multinode-aware version of a process registry.

When you attempt to register a global alias, a cluster-wide lock is set, preventing any competing registration on other nodes. Then the check is performed to see whether the alias is already registered. If not, all nodes are informed about the new registration. Finally, the lock is released. Obviously, this involves a lot of chatter, and multiple small messages are passed between nodes.

Note that lookups are local. When a registration is being performed, all nodes are contacted, and they cache the registration information in their local ETS tables. Each subsequent lookup on any node is performed on that node, without any additional chatter. This means a lookup can be performed quickly, whereas registration requires chatting between nodes.

Take a special look at the shape of this pid: `#PID<7954.59.0>`. The first number in the pid string representation isn't 0: this indicates that you're dealing with a process from some other node.

### Recognizing remote processes

It should be obvious by now that a pid identifies both a local and a remote process. In almost all cases, you don't need to worry about the physical location of a process. But it's worth mentioning some network-specific details about pids.

All the pids you've seen up to now have had a similar form: `<0.X.0>`, where `X` is a positive integer. Internally, each process has a node-wide unique identifier. This identifier can be seen in the last two

numbers of the string representation. If you create enough processes on a single node, the third number will also be greater than zero.

The first number represents the node number—an internal identifier of the node where the process is running. When this number is zero, the process is from the local node. Conversely, when output includes a pid in the form `<X.Y.Z>` and `X` isn't zero, you can be sure it's a remote process.

Global registration allows you to forward all requests that need to manipulate the same resource (in this case, a to-do list) to a single synchronization point (a process) in your cluster. This is exactly the same pattern you use in a single-node setting, now applied to a cluster of multiple nodes. You'll see this in action a bit later, when you start making your to-do system distributed.

Global registration can also be used with `GenServer`, as illustrated in the following snippet:

```
GenServer.start_link(__MODULE__, arg,
  name: {:global, some_global_alias}
)

GenServer.call({:global, some_global_alias}, ...)
```

Registers the process under a global alias

A global alias can be used to make a request.

Finally, if a registered process crashes or the owner node disconnects, the alias is automatically unregistered on all other machines.

**Groups of processes**

Another frequent discovery pattern occurs when you want to register multiple processes under the same alias. This may sound strange, but it's useful in situations where you want to categorize processes in the cluster and broadcast messages to all the processes in a category.

For example, in redundant clusters, you want to keep multiple copies of the same data. Having multiple copies allows you to survive node crashes. If one node terminates, a copy should exist somewhere else in the cluster.

For this particular problem, you can use the strangely named `:pg2` (process groups, version 2) module (http://erlang.org/doc/man/pg2.html). This module allows you to create arbitrarily named cluster-wide groups and add multiple processes to those groups. This addition is propagated across all nodes, and, later, you can query the group and get the list of all processes belonging to it.

Let's try this. You'll set up both shell processes of `node1` and `node2` to handle Bob's to-do list. To do this, you need to

1. Create a group of processes for Bob's list.
2. Add both processes to this group.

Creating a process group is as simple as calling `:pg2.create/1` on any node and providing an arbitrary term that serves as the group's identifier. Let's do this on `node1`:

```
iex(node1@localhost)11> :pg2.start

iex(node1@localhost)12> :pg2.create({:todo_list, "bob"})
:ok
```

This group is immediately visible on `node2`:

```
iex(node2@localhost)9> :pg2.start

iex(node2@localhost)10> :pg2.which_groups
[todo_list: "bob"]
```

Remaining on `node2`, you can now add the shell process to this group:

```
iex(node2@localhost)11> :pg2.join({:todo_list, "bob"}, self)
:ok
```

This change should be visible on `node1`:

```
iex(node1@localhost)13> :pg2.get_members({:todo_list, "bob"})
[#PID<8531.59.0>]
```

Finally, you can add the `node1` shell process to the same group:

```
iex(node1@localhost)14> :pg2.join({:todo_list, "bob"}, self)
:ok
```

At this point, both processes are in the process group, and both nodes can see this:

```
iex(node1@localhost)15> :pg2.get_members({:todo_list, "bob"})
[#PID<8531.59.0>, #PID<0.59.0>]

iex(node2@localhost)12> :pg2.get_members({:todo_list, "bob"})
[#PID<0.59.0>, #PID<7954.59.0>]
```

How can you use this technique? When you want to make an update to Bob's to-do list, you can query the corresponding process group and get a list of all processes responsible for Bob's list. Then you can issue your request to all processes: for example, by using `GenServer.multi_call/4`. This ensures that all replicas in the cluster are updated.

But when you need to issue a query (for example, retrieve to-do list entries), you can do this only on a single process from the group (no need to perform multiple queries on all replicas, unless you want better confidence). Therefore, you can choose a single pid from the process group. For this purpose, you can use `:pg2.get_closest_pid/1`, which returns the pid of a local process, if one exists, or a random process from the group otherwise.

Just like the `:global` module, `:pg2` is implemented in pure Erlang and is also an elaborate version of a process registry. Group creations and joins are propagated across the cluster, but lookups are performed on a locally cached ETS table. Process crashes and node disconnects are automatically detected, and nonexistent processes are removed from the group.

### 12.1.4. Links and monitors

Links and monitors work even if processes reside on different nodes. A process receives an exit signal or a `:DOWN` notification message (in the case of a monitor) if any of the following events occurs:

- Crash of a linked or monitored process
- Crash of a BEAM instance or the entire machine where the linked or monitored process is running
- Network connection loss

Let's quickly prove this. You'll start two nodes, connect them, and set up a monitor from the `node1` shell to the shell of `node2`:

```
$ iex --sname node1@localhost

$ iex --sname node2@localhost
iex(node2@localhost)1> Node.connect(:node1@localhost)
iex(node2@localhost)2> :global.register_name({:todo_list, "bob"}, self)
iex(node1@localhost)1> Process.monitor(
  :global.whereis_name({:todo_list, "bob"})
)
```

Monitors a process on another node

Now you can terminate `node2` and flush messages in `node1`:

```
iex(node1@localhost)2> flush
{:DOWN, #Reference<0.0.0.99>, :process, #PID<7954.59.0>,
:noconnection}
```

As you can see, you have a notification that the monitored process isn't running anymore. This allows you to detect errors in distributed systems and recover from them. In fact, the error-detection mechanism works the same as in concurrent systems, which isn't surprising given that concurrency is also a distribution primitive.

This can be useful in all kinds of situations where you depend on a process from a remote node. For example, when a `GenServer.call` is issued, the underlying implementation first sets up a monitor to the receiver process and then sends a request message. Consequently, the caller receives either a response message or a `:DOWN` message.

### 12.1.5. Other distribution services

Other interesting services are provided as part of the Erlang standard library. I'll mention them briefly here, but once you start writing distributed systems, you should definitely spend time researching them.

I already mentioned that many basic primitives can be found in the `Node` module (http://elixir-lang.org/docs/stable/elixir/Node.html). On top of that, you may find some useful services in the `:net_kernel` (http://erlang.org/doc/man/net_kernel.html) and `:net_adm`(http://erlang.org/doc/man/net_adm.html) modules.

Some additional services are available. Occasionally you'll need to issue function calls on other nodes. As you've seen, this can be done with `Node.spawn`, but this is a low-level approach and often isn't suitable. The problem with `Node.spawn` is that it's a fire-and-forget kind of operation, so you don't know anything about its outcome.

More often, you'll want to obtain the result of a remote function call or invoke a function on multiple nodes and collect all the results. In such cases, you can refer to the `:rpc` Erlang module (http://erlang.org/doc/man/rpc.html), which provides various useful helpers. For example, to call a function on another node and get its result, you can use `:rpc.call/4`, which accepts a node and an MFA identifying the function to be called remotely. Here's an example that performs a remote call of `Kernel.abs(-1)` on `node2`:

```
iex(node1@localhost)1> :rpc.call(:node2@localhost, Kernel, :abs,
[-1])
1
```

Other useful helpers included in the `:rpc` module allow you to issue a remote function call on multiple nodes in the cluster. You'll see this in action a bit later, when you add replication feature to your database.

I also want to mention cluster-wide locks. These are implemented in the `:global` module and allow you to grab an arbitrary named lock. Once you have a particular lock, no other process in the cluster can acquire it until you release it.

**Message passing is the core distribution primitive**

Many services, such as `:rpc`, are implemented in pure Erlang. Just like `:global` and `:pg2`, `:rpc` relies on transparent message

passing and the ability to send messages to locally registered processes on remote nodes. For example, `:rpc` relies on the existence of a locally registered `:rex` process (which is started when Erlang's `:kernel` application is started). Making an rpc call on other nodes amounts to sending a message containing MFA to `:rex` processes on target nodes, calling `apply/3` from those servers, and sending back the response.

If you want to dive deeper into distributed programming on Erlang systems, spend some time studying the code for `rpc.erl`, `pg2.erl`, and `global.erl` to learn about various distributed idioms and patterns.

Let's see this in action. Start `node1` and `node2` and connect them. Then, on `node1`, try to acquire the lock using `:global.set_lock/1`:

```
iex(node1@localhost)1> :global.set_lock({:some_resource, self})
true
```

The tuple you provide consists of the resource ID and the requester ID. The resource ID is an arbitrary term, whereas the requester ID identifies a unique requester. Two different requesters can't acquire the same lock in the cluster. Most often, you'll want to use the process ID as the requester ID, which means that at any point, at most one process can acquire the lock.

Acquiring the lock involves chatting with other nodes in the cluster. Once `:set_lock` returns, you know that you have the lock, and no one else in the cluster can acquire it. Let's attempt to acquire the lock on `node2`:

```
iex(node2@localhost)1> :global.set_lock({:some_resource, self})
```

Blocks until the lock is released

The shell process on `node2` will wait indefinitely (this can be configured via an additional parameter) until the lock becomes available. As soon as you release the lock on `node1`, it's obtained on `node2`:

```
iex(node1@localhost)2> :global.del_lock({:some_resource, self})
iex(node2@localhost)2>
```

The lock is now held by the shell process on node2

There is also a simple helper for the acquire/release pattern available in the form of `:global.trans/2` (http://erlang.org/doc/man/global.html#trans-2), which takes the lock, then runs the provided lambda, and finally releases the lock.

Locking is something you should usually avoid, because it causes the same kinds of problems as classical synchronization approaches. Excessively relying on locks increases the possibility of deadlocks, livelocks, or starvation. Most often, you should synchronize through processes, because it's easier to reason about the system this way.

But, used judiciously, locks can sometimes improve performance. Remember that message passing has an associated cost; this is especially true in distributed systems, where a message must be serialized and transmitted over the network. If a message is very large, this can introduce significant delays and hurt system performance.

Locks can help here, because they let you synchronize multiple processes on different nodes without needing to send large messages to another process. Here's a sketch of this idea. Let's say you need to ensure that the processing of a large amount of data is serialized in the entire cluster (at any point in time, at most one process may run in the entire cluster). Normally, this is done by passing the data to a process that acts as a synchronization point. But passing a large chunk of data may introduce a performance penalty because data must be copied and transmitted over the network. To avoid this, you can synchronize different processes with locks and then process the data in the caller context:

```
def process(large_data) do
  :global.trans(                          <-- Acquires the cluster-wide lock
    {:some_resource, self},
    fn ->
      do_something_with(large_data)       <-- Runs in the caller process
    end
  )
end
```

Calling `:global.trans/2` ensures cluster-wide isolation. At most one process in the cluster can be running `do_something_with/1` on `:some_resource` at any point in time. Because `do_something_with/1` is running in the caller process, you avoid sending a huge message to another synchronization process. Invoking `:global_trans/2` introduces additional chatter between nodes; but messages used for acquiring the lock are much smaller compared to passing the contents of `large_data` to another process on another node, so you save bandwidth.

This concludes the discussion of the basics of distribution. I didn't mention some important aspects that arise once you start using a network as a communication channel for message passing. We'll revisit this topic in the last section of this chapter; but for now, let's focus on making your to-do system more distributed.

## 12.2. Building a fault-tolerant cluster

With some distribution primitives in your arsenal, you can begin building a cluster of to-do web servers. The aim is to make your system more resilient to all sorts of outages, including crashes of entire nodes. The solution presented here is course overly simplistic. Making a proper distributed system requires much more attention to various details, and the topic could easily fill an entire book.

On the plus side, making a basic BEAM-powered distributed system isn't complicated. In this section, you should get a feel for how distribution primitives fit nicely into the existing BEAM concurrency model.

Most of your work here will be based on the `GenServer` abstraction. This shouldn't come as a surprise, given that message passing is the main distribution tool in BEAM. So before continuing, make sure you remember how `GenServer` works; if needed, revisit the explanation in chapter 6.

### 12.2.1. Cluster design

The goals of this cluster are deceptively simple:

1. The cluster will consist of multiple nodes, all of which are powered by the same code and provide the same service (a web interface for managing multiple to-do lists).

2. Changes should be propagated across the cluster. A modification made on a single to-do list on one node should be visible on all other nodes. From the outside, clients shouldn't care which node they access.

3. The crash of a single node shouldn't disturb the cluster. Service should be provided continuously, and data from the crashed node shouldn't be lost.

These goals describe a fault-tolerant system. You always provide service, and individual crashes don't cause a disturbance. Thus the system becomes more resilient and highly available.

**Network partitions**

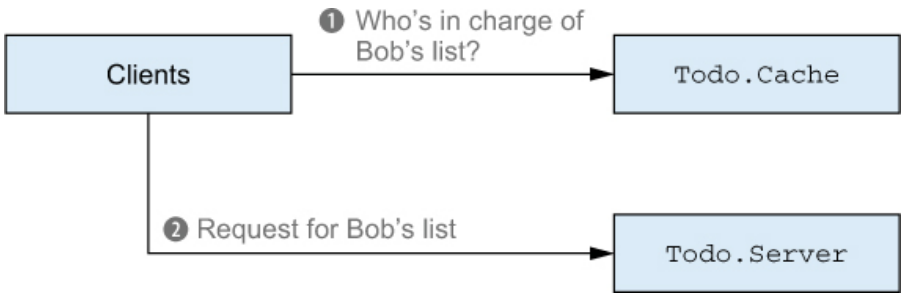
Note that you won't tackle the most difficult challenge of distributed systems: network partitions. A *partition* is a situation in which a communication channel between two nodes is broken and the nodes are disconnected. In this case, you may end up with a "split brain" situation: the cluster gets broken into two (or more) disconnected smaller clusters, all of which work and provide service. This situation can cause problems because now you have multiple isolated systems, each of which accepts input from users. Ultimately, you may end up with conflicting data that is impossible to reconcile. For most of this section, we'll ignore this issue, but we'll discuss some consequences before parting.

Let's begin work on making your system distributed. First, we'll look at the to-do cache.

### 12.2.2. The distributed to-do cache

In a sense, the to-do cache is the centerpiece of your system. This is the primary element that maintains the consistency of your model, so let's recall how it works. The main idea is illustrated in figure 12.2.

**Figure 12.2. Using a to-do cache**




When you want to modify a to-do list, you ask the to-do cache to provide the corresponding to-do server process for you. This to-do server then acts as a synchronization point for a single to-do list. All requests for Bob's list go through that process, which ensures consistency and prevents race conditions.

When clustering your system, you'll aim to keep this property. The difference is only that your cache must somehow be made to work across all nodes in the cluster. So, wherever in the cluster you ask the question "Who is in charge of Bob's list?" the answer always points to the same process in the cluster (until that process crashes, of course). This is the single thing you need to change to make your to-do cache distributed; and as you'll see, the changes are reasonably straightforward.

#### Discovering to-do servers

There are various ways of doing cluster-wide discovery. Probably the simplest (although not necessarily the most efficient) relies on services from the `:global` module that allow you to register a process under a global alias—an arbitrary term that identifies a process in the cluster. Here's what you need to do:

1. Adapt the `Todo.Server` module to use global registration.
2. Adapt `Todo.Cache` to work with the new registration.

Let's start implementing this. The first thing you'll do is modify the `Todo.Server` module to rely on global registration. So far, you've been using the `gproc` application as a registration facility. But `gproc` is most suitable for single-node registrations; although it also supports cluster-wide registration, this feature relies on additional libraries, and it's definitely not the simplest approach. In this case you'll opt for the `:global` module.

**Process registrations**

You may be puzzled by all these different registration facilities, so let's recall the key differences. The basic registration facility is a local registration that allows you to use a simple atom as an alias to the single process on a node. `gproc` extends this by letting you use rich aliases—any term can be used as an alias. `gproc` is more suitable for single-node registrations, although it does provide support for global registrations as well.

Reaching for `:global` allows you to register a cluster-wide alias. Finally, `:pg2` is useful to register multiple processes behind a cluster-wide alias (process group), which is usually suitable for pub-sub scenarios.

Replacing `gproc` with `:global` is fairly simple. The changes are presented in the following listing.

**Listing 12.1. Global registration of to-do servers (todo_distributed/lib/todo/server.ex)**

```
defmodule Todo.Server do
  ...
  def start_link(name) do
    IO.puts "Starting to-do server for #{name}"
    GenServer.start_link(
      Todo.Server, name,
      name: {:global, {:todo_server, name}}          <-- Global registration
    )
  end

  ...

  def whereis(name) do
    :global.whereis_name({:todo_server, name})       <-- Global discovery
  end

  ...
end
```

Nothing particular fancy is happening here. You replace `gproc` and use `:global`, relying on the built-in integration with `GenServer` that allows you to use the `name: {:global, alias}` option when creating the process.

In addition, you use `:global.whereis_name/1` to discover the process. As already mentioned, this function doesn't perform a cluster-wide search; instead, it relies on a local ETS table managed by the `:global` module. When you register a process under a global alias, the `:global` module performs a chat across the entire cluster, and this registration is propagated to all nodes. If a new node joins the cluster, it also exchanges information with other nodes, and global registration caches are updated. Likewise, if the host node disconnects from the cluster in any way (explicitly, by crash, or by network partition), or the registered process crashes, the alias is unregistered on all nodes in the cluster.

Next, you need to change the to-do cache process. Let's recall how this process works. Whenever an arbitrary client needs to obtain a to-do server pid, it calls `Todo-Cache.server_process/1`. In this function, you do the following:

**1**. Perform a lookup to find a to-do server process, and return its pid if it exists. This lookup is performed in the client's process.

**2**. If the process isn't found, issue a call to the cache process.

**3**. In the cache process, perform the lookup again, returning the pid if it exists. This extra lookup prevents race conditions.

**4**. If the to-do server doesn't exist, create the new one (via a call to `Todo.ServerSupervisor.start_child/1`) and return its pid.

Due to the way global registration works, coupled with its integration to `GenServer` and `Supervisor`, you can simplify this process. In particular, a couple of nice properties can help you:

- I mentioned earlier that global registration sets a cluster-wide lock, and the registration is done in the synchronized piece of

code. At any point, there can be at most one process in the entire cluster performing the registration. Therefore, you don't need to synchronize the registration code yourself. You can safely call multiple simultaneous global registrations of the same alias, and the first one will win; other processes competing for the same alias won't succeed.

- When you use `GenServer.start_link` with the `name: {:global, {:todo_server, name}}` option, if the registration fails because another process is already registered under this alias, the new process will be terminated. The result of `start_link` will be in the form `{:error, {:already_started, pid}}`, where `pid` identifies the process registered under this alias.
- If you use `Supervisor.start_child` to start a `GenServer`, and `GenServer .start_link` returns an error, `start_child` propagates this error to you, and you can do something about it.

Consequently, when you attempt to start the to-do server process by calling `Todo .Supervisor.start_child/1`, you can expect the following results:

- `{:ok, pid}`—The new process has been created and registered.
- `{:error, {:already_started, pid}}`—Another process is already registered under this alias, and the new process has been terminated.

This means you can safely attempt to start the to-do server without synchronizing the operation through the single cache process. If two registrations of the same to-do list happen to run simultaneously, one will succeed and the other one will return an error with the pid of the process that registered successfully. In both cases, you'll know the pid of the authority for the given to-do list, and you can use it to interact with the list.

Consequently, the `Todo.Cache` module can be significantly simplified. It doesn't need to implement a `GenServer`, and a process discovery can be performed in the client process. The full implementation is provided in the next listing.

**Listing 12.2. Locking synchronization in the cache (todo_distributed/lib/todo/cache.ex)**

```
defmodule Todo.Cache do
  def server_process(todo_list_name) do
    case Todo.Server.whereis(todo_list_name) do
      :undefined -> create_server(todo_list_name)
      pid -> pid
    end
  end

  defp create_server(todo_list_name) do
    case Todo.ServerSupervisor.start_child(todo_list_name) do
      {:ok, pid} -> pid
      {:error, {:already_started, pid}} -> pid
    end
  end
end
```

Verifies whether the process already exists

A new process is successfully created.

Another process has already been registered.

Notice that you first check to see whether the process is already registered. Theoretically, you could avoid this check, but then every attempt to access the existing to-do server would be less efficient, because it would involve node chatting in the attempt to register the process.

By looking up whether the process is already there, you can reduce the node chatter. Recall that a lookup is done locally in an internal ETS table. This means you can quickly verify whether the process exists and avoid chatting with other nodes if the process is already registered.

#### Alternative discovery

Keep in mind that global registration is chatty and serialized (only one process at a time may perform global registration). This means the approach you used isn't very scalable with respect to the number of different to-do lists or the number of nodes in the cluster. The solution will also perform poorly if the network is slow.

Of course, there are alternatives. The main challenge here is to reliably discover the process responsible for a to-do list while reducing network communication. This can be done by introducing a rule that always maps the same to-do list name to the same node in the network. Here's a simple sketch of the idea:

```
def node_for_list(todo_list_name) do
  all_sorted_nodes = Enum.sort(Node.list([:this, :visible]))

  node_index = :erlang.phash2(
```

```
    todo_list_name,
    length(all_sorted_nodes)
  )

  Enum.at(all_sorted_nodes, node_index)
end
```

You get the list of all nodes and sort it to ensure that it's always in the same order. Then you hash the input name, making sure the result falls in the range `0..length(all_sorted_nodes)`. Finally, you return the node at the given position. This ensures that as long as the cluster is stable (the list of nodes doesn't change), the same to-do list will always be mapped to the same node.

Now you can make a discovery in a single hop to the single node. Assuming the previous version of `Todo.Cache` (not the one you just implemented), retrieving the target process can be as simple as this:

```
node_for_list(todo_list_name)
|> :rpc.call(Todo.Cache, :server_process, [todo_list_name])
```

You forward to the target node and retrieve the desired process there. You don't need to use global registration, and `Todo.Cache` can continue working as it was until this chapter. The result of the previous invocation is a pid, which you can then use to make your call. The benefit is that you can discover the pid with less chatting.

The main downside of this approach is that it doesn't work properly when the cluster configuration changes. If you add another node or a node disconnects, the mapping rules will change. Dealing with this situation is complex. You need to detect the change in the cluster (which is possible, as explained a bit later) and migrate all data to different nodes according to new mapping rules. While this data is being migrated, you'll probably want to keep the service running, which will introduce another layer of complexity. The amount of data that needs to be migrated can be greatly reduced if you use some form of consistent hashing (http://en.wikipedia.org/wiki/Consistent_hashing)—a smarter mapping of keys to nodes, which is more resilient to changes in the cluster.

It's obvious that the implementation can quickly become more involved, which is why we started simple and chose the global

registration approach. Although it's not particularly scalable, it's a simple solution that works. But if you need better performance and scale, you'll have to resort to a more complex approach.

### 12.2.3. Implementing a replicated database

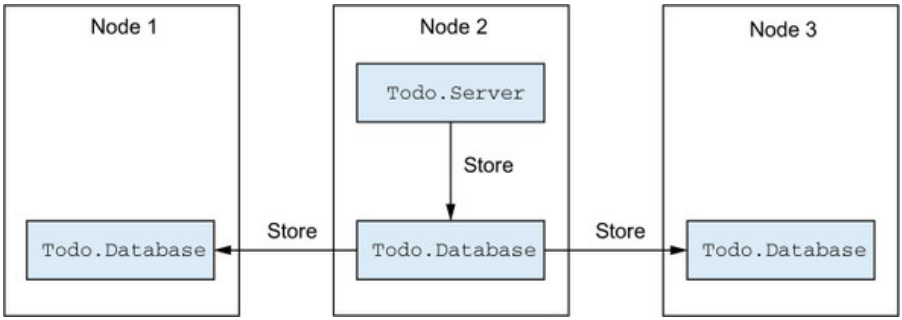
After the changes you just made, you'll have the following behavior:

1. When the first request for Bob's list arrives, a to-do list is created on the node that handles that request.
2. All subsequent requests on Bob's to-do list are forwarded to the process created in step 1.
3. If the node (or the process) created in step 1 crashes, a new request for Bob's list will cause the new to-do server to be registered.

Everything seems fine at first glance, and the system looks properly distributed. You won't test it now because there is one important issue you haven't addressed yet: the database doesn't survive crashes. Let's say you performed multiple updates to Bob's list on node A. If this node crashes, some other node, such as node B, will take over the work for Bob's list. But previously stored data won't be on that node, and you'll lose all your changes.

Obviously you need to make your database replicated, so that data can survive node crashes. The simplest (although not the most efficient) way of preserving data is to replicate it in the entire cluster. This idea is illustrated in figure 12.3.

**Figure 12.3. Replicating the database**




This is pretty straightforward. When you store data to the database, you'll propagate the change to all nodes in the cluster. The corresponding implementation can be simple if you rely on services from the `:rpc` module. I mentioned that `:rpc`, among other things, allows you to issue a function call on all nodes in the cluster. You'll rely on this feature and make some changes to your `Database` module:

1. You'll rename the existing `Database.store` function to `Database.store_local`. The code remains the same.

2. You'll provide the new implementation to `Database.store`. This new implementation will call `Database.store_local` on all nodes in the cluster.

Another change you'll make (the code isn't presented here) will turn `Todo.Database-Worker.store/2` into a call. This request should have been implemented as a call in the first place. Back in chapter 7, I opted for a cast somewhat arbitrarily, mostly for didactic purposes. In reality, if you're asking another process to store data, you should request a confirmation message so you know whether the data has been stored or something went wrong. This becomes increasingly important as you move to a less reliable communication medium (network) where all sorts of things can go wrong. When you expect a request to be used across nodes, you should usually implement it as a call.

No other changes are required. In particular, `Todo.Database.get/1` remains unchanged. When you want to read the data, you'll do it from the local node, optimistically assuming that

all nodes have the same copy of the data. The changes are pretty simple, as shown in the following listing.

**Listing 12.3. Storing data on all nodes (todo_distributed/lib/todo/database.ex)**

```
defmodule Todo.Database do
  ...
  def store(key, data) do
    {results, bad_nodes} =
      :rpc.multicall(
        __MODULE__, :store_local, [key, data],
        :timer.seconds(5)
      )

    Enum.each(bad_nodes, &IO.puts("Store failed on node #{&1}"))

    :ok
  end

  ...
end
```

Calls store_local on all nodes

Logs failed results

Here, you rely on `:rpc.multicall/4` to make a function call on all nodes in the cluster. `multicall` accepts MFA and a timeout. The target function is then invoked on all nodes in the cluster, all results are collected, and you get a tuple in the form `{results, bad_nodes}`: a list of results and a list of nodes that didn't respond in the given time.

### Always provide timeouts

The timeout provided to `multicall` is important. Without it, `multicall`, and in turn your `store` operation, would be blocked forever.

When doing distributed calls, you should usually specify a timeout. Note that this is no different from cross-process calls—when making a call to another process, you usually want to have a timeout as well, and `GenServer` recognizes this by providing a default 5-second timeout. Again, cross-node operations aren't all that different from cross-process operations, and in many cases you have to consider a similar set of problems when developing a single-node system.

Finally, you print all nodes on which the request timed out. Note that

in practice, this isn't sufficient. You should also verify that each received response returns `:ok`. Moreover, you should do something meaningful in the case of a partial success. Otherwise, you'll end up with an inconsistent cluster, with different nodes containing different data. To deal with this situation properly, you need to implement a two-phase commit protocol and a log-based database implementation. This will allow you to safely roll back in case of a partial failure. For the sake of simplicity and brevity, I refrain from doing this here, but in a real project this is an issue that needs to be considered and addressed.

I made another small change in the database workers, which isn't presented in the code. Up to now, you've used the persist folder to store your data. This is changed to accommodate the node name. So, if your node is called `node1@localhost`, you'll store data in the persist/node1 folder. This is done mostly to simplify testing and to allow you to start multiple nodes locally from the same root folder.

In any case, this simple change makes it possible to replicate your data across the cluster. With this, your basic take on a clustered to-do system is finished, and you can try it in action.

### 12.2.4. Testing the system

Finally it's time to test the system. You need to start a few nodes, connect them, and see how the cluster works. But recall that in chapter 11, you made the web server listen on port 5454. You can't have two nodes listening on the same port, so you need to change this. Luckily, you made the web port configurable via the application environment, so it's possible to change the default port from the command line.

Start two instances, `node1` and `node2`, that listen on ports 5454 and 5555, respectively:

```
$ iex --sname node1@localhost -S mix
$ iex --erl "-todo port 5555" --sname node2@localhost -S mix
```

Starts node1, which listens on the default port

Starts node2, and sets the alternative port

Next, you need to connect the two nodes:

```
iex(node1@localhost)1> Node.connect(:node2@localhost)
```

Now the cluster is established, and you can use your servers. Add an entry for Bob on the first node:

```
$ curl -d "" \
  "http://localhost:5454/add_entry?
list=bob&date=20131219&title=Dentist"
OK
```

Then, verify if this entry is visible on another node:

```
$ curl "http://localhost:5555/entries?list=bob&date=20131219"
2013-12-19    Dentist
```

This proves that your data is propagated across the cluster. Furthermore, looking at individual `iex` shells, you see the "Starting to-do server for bob" message in the `node1` shell but not in `node2`. This is clear proof that even when you try to access Bob's list on another node, you're forwarded to the corresponding process on `node1`.

You can thus safely modify Bob's list on `node2` without compromising the data:

```
$ curl -d "" \
  "http://localhost:5555/add_entry?
list=bob&date=20131219&title=Movies"

$ curl "http://localhost:5454/entries?list=bob&date=20131219"

2013-12-19    Movies
2013-12-19    Dentist
```

Finally, crashing a single node won't disturb the system. Let's stop `node1`, where Bob's to-do server is running, and try to query `node2`:

```
$ curl "http://localhost:5555/entries?list=bob&date=20131219"
2013-12-19    Movies
2013-12-19    Dentist
```

Sure enough, the cluster is still providing its service, and data is

preserved. The new to-do server has been created on `node2`, and it restored the state from the replicated database.

At this point, your basic cluster is complete. There are some remaining issues, which I don't address here but will mention:

- You should set up a load balancer to serve as a single access point for all clients.
- You don't have a scheme for introducing new nodes to the running cluster. When a new node is introduced, it should first synchronize the database with one of the already-connected nodes; then it can begin serving requests.
- Database replication is fragile. You need some kind of two-phase commit strategy.
- You don't deal with network partitions.

Some of these challenges aren't easy to tackle, but they're inherent to distributed systems, and you'll have to deal with them regardless of the underlying technology. It's important to understand that Erlang isn't a magic wand for distributed problems. In a distributed system, many things can go wrong, and it's up to you to decide how you want to recover from various failures. There is no one-size-fits-all solution: your job is to combine basic distribution primitives in a way that suits the problem at hand.

Of course, offloading the work to proven third-party components can often help. For example, by using the built-in Mnesia database, you could achieve better write guarantees and be able to easily migrate new nodes to the cluster. But even then, it's important to understand how a third-party library works in a distributed setting. In this example, Mnesia doesn't deal explicitly with network partitions and split-brain scenarios, and instead leaves it to the developer to resolve this situation. Some other component might exhibit different drawbacks, so you need to understand how it works in a distributed setting.

Erlang distribution primitives can take you a long way. You've made it pretty far, and only a few changes were needed to make your system distributed, even if you didn't prepare for the distributed system up front.

### 12.2.5. Dealing with partitions

The work so far has been easy, but we've conveniently ignored the issue of network partitions. This is one of the biggest challenges when building a distributed system. Fully discussing this topic could easily turn into a substantially sized book, so here I'll provide just a basic overview.

When you decide to go distributed, partitions are a problem you'll have to deal with, one way or another. Even if you reach for a third-party product (such as an external database) to handle clustering and replication, you should understand how that product behaves when partitions occur. A network partition is a situation you shouldn't ignore in a distributed system, and it's best to be aware of the challenges you'll face so you can make a conscious and an informed decision about how to proceed.

A network partition, or *netsplit*, is a situation in which two nodes can no longer communicate with each other. There can be all sorts of underlying causes, and it's impossible to tell them apart:

- A network connection is lost.
- A network connection is extremely slow.
- A remote node has crashed.
- A remote node is overloaded and busy to the point that it can't respond in a timely manner.

From the standpoint of one node, all those situations look the same. The remote node doesn't respond, and you don't know why. It's therefore virtually impossible to guarantee that a netsplit will never take place. Even on an ultra-fast and reliable network, a bug or overload may cause one host to become so busy that it can't respond to the other in a timely manner. The other node has no choice but to interpret this situation as a netsplit and conclude that the connection is lost. This means when you're implementing a distributed system, you need to consider network partitions and devise a strategy to deal with such situations.

When a partition occurs, you may end up with multiple independent

clusters that are mutually disconnected. The problem is that although those clusters can't talk to each other, a cluster's clients may be able to reach all nodes. As mentioned earlier, this situation is also known as *split-brain*. If different clusters continue to serve users independently, you may end up with undesired behavior. Issuing a request on one cluster won't be visible on another, and users may face lost updates, or phantom entries may appear. Ultimately, once you reconnect those clusters, you may end up with conflicting data.

**The CAP theorem**

Dealing with partitions requires careful consideration and involves making some tradeoffs. This is more formally known as the CAP theorem. CAP stands for the following:

- ***Consistency—*** When a write is performed, all subsequent reads will see this write (until the next write is performed).
- ***Availability—*** All nodes running in the system can provide full service, even if they can't talk to each other. Each request must succeed (assuming the input is valid).
- ***Partition tolerance—*** The system can provide service during network partitions.

According to the theorem (which is formally proven), during a network partition, you can have either consistency (a CP system) or availability (an AP system), but never both.

A CP system strives to preserve data consistency, even if it means you stop providing some part of the service during the partition. For example, in a banking system, you may choose to deny service during the partition rather than risk compromising the consistency of bank accounts.

A reasonably simple way of making a CP system is to require a quorum: that is, a majority of nodes must be present in order for the cluster to function. For example, let's say you're running a cluster of five nodes, and you replicate all data across all nodes. You can then say that you need at least three nodes to function properly. When a netsplit occurs (which can be detected as described a little later), each node can check how large the cluster is that it belongs to. The nodes

that are in the majority continue working, whereas nodes in the minority stop providing service. In more formal terms, if you want to tolerate F number of disconnects (or node failures), you need to have at least 2F + 1 nodes in the cluster. For example, in a cluster of 7 nodes, if more than 3 nodes disconnect, you'll stop providing service.

In an AP system, you take the opposite route and aim for availability at the cost of consistency. For example, in an online shop, it might be important to accept as many purchases as possible even if that means making an occasional false promise, such as selling an item you no longer have in stock.

An AP system can continue working even when a full split occurs and each node runs on its own. You may end up with divergent changes—people may modify the same data on different nodes, and this can lead to conflicts. Once you rejoin the cluster, you'll need to reconcile those changes. With some smart algorithms, it's possible to establish the logical order of changes and automatically merge nonconflicting ones. Nevertheless, there will be some conflicts, and these will have to be dealt with either using an automatic strategy (such as "last write wins") or by asking a human (an end user or an administrator) to resolve them.

Ultimately, it's up to you to decide which approach to take. An AP system is more responsive and available both in the normal mode of operation and during partitions. A CP system can become less available during partitions but preserves consistency and eliminates the possibility of conflicts.

**Note**

You might think that a system could be CA, but that doesn't make sense. A CA system would offer both consistency and availability as long as there were no network partitions. But this statement holds for any properly developed system. The point of the CAP theorem is to decide how the system should behave during network partitions; and in such cases, you have to choose either consistency or availability. Partition tolerance therefore isn't an option, and you must decide what to do when it happens.

It of course goes without saying that you should try to rely on existing, proven third-party solutions as much as possible, especially when it comes to the database. Using a battle-tested solution instead of rolling a custom one is most often a sensible solution. A good knowledge resource that discusses the netsplit behavior of various products is the excellent "Jepsen" blog series by Kyle Kingsbury (http://aphyr.com/tags/jepsen).

#### Detecting netsplits

As it turns out, detecting a partition in Erlang is simple. Remember, a partition always manifests as a loss of connection to the remote node, and it's possible to detect this situation. As mentioned earlier, a node periodically pings its peers via tick messages, and if a peer fails to respond to these messages, it will be considered disconnected. Each process can subscribe to notifications about changes in connected nodes via `:net_kernel.monitor_nodes/1` (http://mng.bz/9799).

The argument you provide is a boolean that indicates whether you're adding a new subscription (`true`) or installing a single subscriber, overwriting all previous ones on this node (`false`). Either way, a process that calls `monitor_nodes` will receive notifications whenever a remote node connects or disconnects.

Let's try this. First, start `node1` and subscribe to notifications:

```
$ iex --sname node1@localhost
iex(node1@localhost)1> :net_kernel.monitor_nodes(true)
```

This makes the caller process (in this case, the shell) receive notifications. Now, start two additional nodes and connect them to `node1`:

```
$ iex --sname node2@localhost
iex(node2@localhost)1> Node.connect(:node1@localhost)

$ iex --sname node3@localhost
iex(node3@localhost)1> Node.connect(:node1@localhost)
```

In the `node1` shell, you can see the corresponding messages:

```
iex(node1@localhost)2> flush
{:nodeup, :node2@localhost}
```

```
{:nodeup, :node3@localhost}
```

The same thing happens on disconnect. You can stop `node2` and `node3`, and check the messages in `node1`:

```
iex(node1@localhost)3> flush
{:nodedown, :node3@localhost}
{:nodedown, :node2@localhost}
```

Alternatively, you can also use `Node.monitor/2` (http://mng.bz/ge5f) if you want to explicitly set nodes that you want to monitor.

Either way, detecting a partition amounts to listening for `:nodedown` messages. For example, if you want to ensure that you're still part of the majority, you can handle a `:nodedown` message by checking the number of nodes to which you're still connected. If the quorum is met, you continue running; otherwise, you must stop your service—for example, by stopping the application.

Finally, as I already mentioned, you can set up a monitor or a link to a remote process. This works just as it does with local processes. If a remote process crashes (or the node disconnects), you'll receive a message (when using monitors) or an exit signal (when using links).

### 12.2.6. Highly available systems

Way back in chapter 1, I described some properties of a highly available system. It may not be obvious, but you've gradually reached this goal in the to-do system, which now has some nice properties:

- ***Responsiveness—*** Because you have a highly concurrent system, you can use your hardware more efficiently and serve multiple requests concurrently. Owing to how BEAM processes work, you won't experience unexpected pauses, such as system-wide garbage collection (because processes are GC-ed individually and concurrently). Occasional long-running tasks won't block the entire system, due to frequent preemption of processes. Ultimately, you should have a predictable running system with fairly constant latency that degrades gracefully if your system

becomes overloaded.
- ***Scalability—*** Your system is both concurrent and distributed, so you can address increased popularity and load by using a more powerful machine or by adding more nodes to the system. The system can automatically take advantage of the new hardware.
- ***Fault-tolerance—*** Due to process isolation, you can limit the effect of individual errors. Due to process links, you can propagate such errors across the system and deal with them. Supervisors can help the system self-heal and recover from errors. At the same time, the main code will follow the happy path, focusing on the work that needs to be done and liberated from error-detection constructs. Finally, due to distribution, you can survive crashes of entire machines in the system.

At this point, it should be clear that the main tool for high availability is the BEAM concurrency model. Relying on processes provided many nice properties and made it possible to come close to having a proper highly available system.

Of course, the system is extremely simplified: you don't provide proper implementations for aspects such as the database, and you don't deal with netsplits, which makes these claims overconfident. Regardless, when you set out to implement a highly available system that must serve a multitude of users continuously, these are the properties you'll need to achieve, and processes are the main tool that can take you there.

At this point, you're finished making your system distributed. Before departing, there are some important network-related considerations to discuss.

## 12.3. Network considerations

So far, you've been running nodes locally. This is fine for making local experiments and doing development time testing. But in production, you'll usually want to run different nodes on different machines. When running a cross-host cluster, you need to consider some additional details. Let's start with node names.

### 12.3.1. Node names

The names you've been using so far are *short names* that consist of an arbitrary name prefix (`node1` and `node2`, in this case) and the host name (`localhost` in these examples). You can also provide a fully qualified node name, also known as *long name*, which consists of a name prefix and a fully qualified host name. A long name can be provided with the `--name` command-line option:

```
$ iex --name node1@127.0.0.1
iex(node1@127.0.0.1)1>
```

Long node name

It's also possible to use symbolic names:

```
$ iex --name node1@some_host.some_domain
iex(node1@some_host.some_domain)1>
```

A node name plays an important role when establishing a connection. Recall that a name uses the form `arbitrary_prefix@host` (short name) or `arbitrary_prefix@ host.domain` (long name). This name obviously identifies a BEAM instance on a machine. The second part of the name (`host` or `host.domain`) must be resolvable to the IP address of the machine where the instance is running. When you attempt to connect to `node2@some_host.some_domain` from `node1`, the `node1` host must be able to resolve `some_host.some_domain` to the IP address of the host machine.

It's also worth noting that a node can connect only to a node that has the same type of name. In other words, a connection between a long-named node and a short-named node isn't possible.

### 12.3.2. Cookies

In order to connect two nodes, they must agree on a magical cookie—a kind of a passphrase that is verified while the nodes are connecting. The first time you start a BEAM instance, a random cookie is generated for you and persisted in your home folder in the file .erlang.cookie. By default, all nodes you start on that machine will have this cookie. To see your cookie, you can use

`Node.get_cookie/0`:

```
iex(node1@localhost)1> Node.get_cookie
:JHSKSHDYEJHDKEDKDIEN
```

Notice that the cookie is internally represented as an atom. A node running on another machine will have a different cookie. Therefore, connecting two nodes on different machines won't work by default; you need to somehow make all nodes use the same cookie. This can be as simple as calling `Node.set_cookie/1` on all nodes you want to connect:

```
iex(node1@localhost)1> Node.set_cookie(:some_cookie)

iex(node1@localhost)2> Node.get_cookie
:some_cookie
```

Another approach is to provide the `--cookie` option when you start the system:

```
$ iex --sname node1@localhost --cookie another_cookie

iex(node1@localhost)1> Node.get_cookie
:another_cookie
```

Cookies provide a bare security minimum and also help prevent a fully connected cluster where all nodes can directly talk to each other. For example, let's say you want to connect node A to B, and B to C, but you don't want to connect A and C. This can be done by assigning different cookies to all the nodes and then, in A and C, using the `Node.set_cookie/2` function, which allows you to explicitly set different cookies that need to be used when connecting to different nodes.

### 12.3.3. Hidden nodes

It should be clear by now that most node operations revolve around the cluster. Most often, you'll treat all connected nodes as being the part of your cluster. But in some cases this isn't what you need. For example, various tools let you connect to the remote running node and interact with it. A simple example is starting a local node that acts as a remote shell to another node. Another example is an

instrumentation tool—a node that connects to another node, collects all sort of metrics from it, and presents the results in a GUI.

Such nodes are helpers that shouldn't be the part of the cluster, and you usually don't want them to be seen as such. For this purpose, you can make a *hidden* connection. When you start your BEAM instance with the `--hidden` argument, the node isn't seen in other nodes' connected lists (and vice versa).

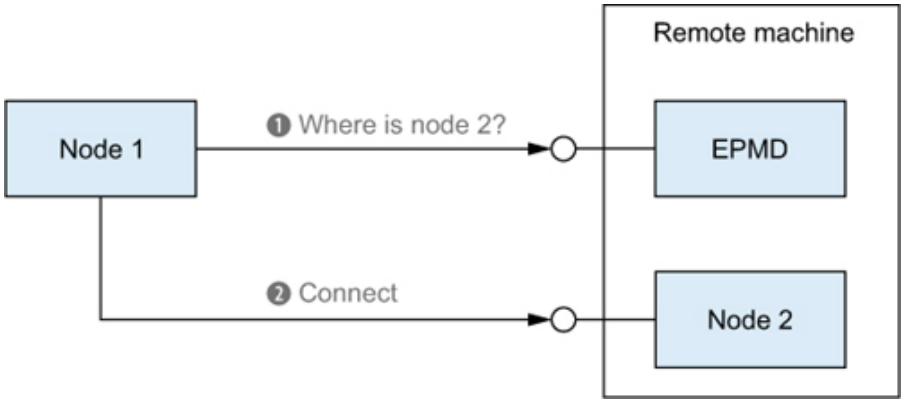
Keep in mind, though, that a hidden node is still maintained in the node's connection list, albeit under a different, `hidden` tag. You can explicitly retrieve hidden nodes by calling `Node.list([:hidden])`. Calling `Node.list([:connected])` returns all connected nodes, both hidden and visible, whereas calling `Node.list([:visible])` returns only visible nodes. When you want to perform a cluster-wide operation, you should generally use the `:visible` option.

Services provided by `:global`, `:rpc`, and `:pg2` ignore hidden nodes. Registering a global alias on one node won't affect any hidden peer, and vice versa.

### 12.3.4. Firewalls

Given that nodes communicate via TCP connection, it's obvious that you need to have some ports that are open to other machines. When one node wants to connect to another node on a different machine, it needs to communicate with two different components, as illustrated in figure 12.4.

**Figure 12.4. Connecting to a remote node**




The first component, the Erlang Port Mapper Daemon (EPMD), is an OS process that's started automatically when you start the first Erlang node on the host machine. This component acts as a node name resolver on the host machine. EPMD knows the names of all currently running BEAM nodes on the machine. When a node wants to connect to a node on this machine, it first queries EPMD to find out the port on which the target node is listening, and then it contacts the target node. EPMD itself listens on the port 4369, and this port must be accessible from remote machines.

In addition, each node listens on a random port, which needs to be accessible as well, because it's used to establish the connection between two nodes. Obviously it's not particularly helpful that the node's listening port is random, because it's not possible to define firewall rules.

Luckily, you can provide a fixed range of ports on which a node will listen. This can be done by setting the `inet_dist_listen_min` and `inet_dist_listen_max` environment variables of the `kernel` app at the command line:

```
$ iex                                                \
  --erl '-kernel inet_dist_listen_min 10000'         \
  --erl '-kernel inet_dist_listen_max 10100'         \
  --sname node1@localhost
```

Sets the range of ports

The node will listen on the first port available in the given range. If you're sure there won't be a port clash, you can use the same value for both parameters, thus effectively designating a single port to be used.

You can manually inspect the ports of all nodes on the host machine via `:net_adm.names/0`:

```
iex(node1@localhost)1> :net_adm.names

{:ok, [{'node1', 10000}]}
```

Alternatively, you can also invoke `epmd -names` from the OS command line. To summarize, if you're behind a firewall, you need to open port 4369 (EPMD) and the range of ports on which your node will listen.

**Security**

Other than the magical cookie, no particular security model is provided. When you connect to a remote node, you can do anything on that node, including running system commands. If the remote node has root privileges, you have a full access to the entire remote host.

Erlang's distributed model was designed to run in a trusted environment, and you should be aware of that fact. In particular, this means in production, your BEAM instances should run under minimal privileges. Moreover, you shouldn't expose your BEAM instances over the internet. If you need to connect nodes from different networks, you should consider switching to SSL as the communication protocol. Some pointers for doing this are provided in the Erlang documentation at www.erlang.org/doc/apps/ssl/ssl_distribution.html.

## 12.4. Summary

In this chapter, you've learned how to make your systems distributed. We covered a lot of ground, and the following points are worth repeating:

- Distributed systems can improve fault tolerance, eliminating the risk of a single point of failure.
- Clustering lets you scale out and spread the total load over multiple machines.

- BEAM-powered clusters are composed of nodes: named BEAM instances that can be connected and can communicate.
- Two nodes communicate via a single TCP connection. If this connection is broken, the nodes are considered disconnected.
- The main distribution primitive is a process. Sending a message works the same, regardless of the process location. A remotely registered process can be accessed via `{alias, node_name}`.
- Building on top of those primitives, many useful higher-level services are available in the `:global`, `:rpc`, and `GenServer` modules.
- When communicating between nodes, use calls rather than casts.
- Always consider and prepare for netsplit scenarios.

You've implemented a distributed, fault-tolerant, scalable to-do HTTP server. The only thing that remains is to prepare the system for production. This is what we'll deal with in the final chapter of the book.

# Chapter 13. Running the system

*This chapter covers*

- Running the system with the mix tool
- OTP releases
- Analyzing system behavior

You spent a lot of time building your to-do system, and it's time to prepare it for production. There are various ways to start your system, but the basic idea is always the same. You have to compile your code as well as your dependencies. Then you start the BEAM instance and ensure that all compiled artifacts are in the load path. Finally, from within the BEAM instance, you need to start your OTP application together with its dependencies. Once the OTP application is started, you can consider your system to be running.

There are various approaches to achieving this, and in this chapter, we'll focus on two of them. First we'll look at how you can use Elixir tools, most notably `mix`, to start the system. Then, we'll discuss OTP releases. Finally, I'll finish the chapter and the book by providing some pointers on how to interact with the running system, which makes it possible to detect and analyze faults and errors that inevitably happen at runtime.

## 13.1. Running a system with the mix tool

Regardless of the method you use to start the system, some common principles always hold. Running the system amounts to doing the following:

1. Compile all modules. Corresponding .beam files must exist somewhere on the disk (as explained in section 2.7).
2. The same holds for the application resource files (.app) of all

OTP applications that are needed to run the system.

3. Start the BEAM instance, and set up load paths to include all locations from steps 1 and 2.

4. Start all required OTP applications.

Probably the simplest way to do this is to rely on standard Elixir tools. Doing so is trivial, and you're already familiar with some aspects of `mix`, `iex`, and `elixir` command-line tools. So far, you've been using `iex`, which lets you start the system and interact with it. When you invoke `iex -S mix`, all the steps just mentioned are taken to start the system.

When running in production, you may want to avoid implicitly starting the `iex` shell. Moreover, you'll probably want to start the system as a background process. Finally, it's beneficial to remove code that exists only for development-time convenience but may otherwise hurt system performance.

Let's start by looking at how you can run the system without the `iex` shell.

### 13.1.1. Bypassing the shell

In chapter 11, I mentioned that it's possible to start the system with `mix run --no-halt`. This command starts the BEAM instance and then starts your OTP application together with its dependencies. The `--no-halt` option instructs `mix` to keep the BEAM instance running forever.

You can achieve the same thing with the `elixir` tool:

```
$ elixir -S mix run --no-halt

Starting database worker 1
Starting database worker 2
Starting database worker 3
```

Starts the system without the iex shell

Although the syntax is a bit more elaborate, `elixir` is more convenient because it's simple to provide command-line arguments for the Erlang virtual machine. This can sometimes be important, because you can make many useful tweaks via command-line arguments. For example, in the current version of Erlang (17), the default maximum number of running processes is set to 262,144. You can increase this by providing the `+P` option:

```
$ elixir --erl "+P 2000000" -S mix run --no-halt
```

Whatever you pass between quotes after the `--erl` option is passed to BEAM. You can find the full list of arguments in the official Erlang documentation (http://erlang.org/doc/man/erl.html). At some point, it's worth spending some time researching the available options. If you want to run a highly concurrent system, you may need to increase some limits and tweak some settings.

### 13.1.2. Running as a background process

Starting the system with `elixir -S mix` gets rid of the shell, but the output is still printed. You can get rid of the output by starting the system in *detached mode*. The OS process will be detached from the terminal, and there will be no console output (it's redirected to `/dev/null`).

Starting a detached BEAM instance is as simple as providing a `--detached` flag to the `elixir` tool. It's also useful to turn the BEAM instance into a node, so you can later interact with it and terminate it when needed:

```
$ elixir --detached --sname foo@localhost -S mix run --no-halt
```

This starts the BEAM instance. You can check that it's running, for example, by looking at which BEAM nodes exist on your system:

```
$ epmd -names

epmd: up and running on port 4369 with data:
name foo at port 51028                    <-------- Foo is running
```

At this point your system is running, and you can use it—for example, by issuing HTTP request to manipulate to-do lists. But how can you stop it?

A useful thing is that you can connect to a running BEAM instance and interact with it. It's possible to establish a *remote shell*—something like a terminal shell session to the running BEAM instance. In particular, you can start another node and use it as a shell to the `foo` node. This can be done using the `--remsh` option:

```
$ iex --sname bar@localhost --remsh foo@localhost --hidden
iex(foo@localhost)1>                    <------ Shell is running on the foo node
```

In this example, you start the `bar` node, but the shell is running in the context of `foo`. Whatever function you call will be invoked on `foo`. This is extremely useful, because now you can interact with the running system. BEAM provides all kinds of nice services that allow you to query the system and individual processes, as we'll discuss a bit later.

Notice that you start the `bar` node as hidden. As mentioned in chapter 12, this means the `bar` node won't appear in the result of `Node.list` (or `Node.list([:this, :visible])`) on `foo`, and therefore it won't be considered part of the cluster.

To stop the running system, you can use the `:init.stop/0` function (www.erlang.org/doc/man/init.html#stop-0), which takes down the system in a graceful manner. It shuts down all running applications and then terminates the BEAM instance:

```
iex(foo@localhost)1> :init.stop
```

The remote shell session is left hanging, and an attempt to run any other command will result in an error:

```
iex(foo@localhost)2>
*** ERROR: Shell process terminated! (^G to start new job) ***
```

At this point, you can close the shell and verify running BEAM nodes:

```
$ epmd -names
```

```
epmd: up and running on port 4369 with data:

$
```

If you want to stop a node programmatically, you can rely on the distributed features described in chapter 12. Here's a quick example:

```
if Node.connect(:foo@localhost) == true do
  :rpc.call(:foo@localhost, :init, :stop, [])
  IO.puts "Node terminated."
else
  IO.puts "Can't connect to a remote node."
end
```

Invokes :init.stop on a remote node

Here, you connect to a remote node and then rely on `:rpc.call/4` to invoke `:init.stop` there.

You can store the code in the file stop_node.exs (the .exs extension is frequently used for Elixir-based scripts). Then you can run the script from the command line:

```
$ elixir --sname terminator@localhost stop_node.exs
```

Running a script starts a separate BEAM instance and *interprets* the code in that instance. After the script code is executed, the host instance is terminated. Because the script instance needs to connect to a remote node (the one you want to terminate), you need to give it a name to turn the BEAM instance into a proper node.

### 13.1.3. Running scripts

This book hasn't discussed the topic of scripts and tools, but they're worth a quick mention. Sometimes you may want to build a command-line tool that does some processing, produces the results, and then stops.

The simplest way to go about it is to write a script. You can create a plain Elixir file, give it an .exs extension to indicate that it's a script, implement module(s), and invoke a function:

```
defmodule MyTool do
  ...

  def run do
    ...
  end
end

MyTool.run                    <-------- Starts the tool
```

You can then invoke the script with the `elixir my_script.exs` command. As a result, all modules you define will be compiled in memory, and all expressions outside of any module will be interpreted. After everything finishes, the script will terminate. Of course, an Elixir script can run only on a system with correct versions of Erlang and Elixir installed.

An .exs script works fine for simpler tools, but it's not efficient when the code becomes more complex, or if you need to include third-party libraries as dependencies. In those cases, it's best to use a proper `mix` project and build a full OTP application.

But because you're not building a system that runs continuously, you also need to include a *runner* module in the project—something that does processing and produces output:

```
defmodule MyTool.Runner do
  def run do
    ...
  end
end
```

Then you can start the tool with `elixir -S mix run -e MyTool.Runner.run`. This starts the OTP application, then invokes the `MyTool.Runner.run/0` function, and terminates as soon as the function is finished.

Finally, you can package the entire tool into an *escript*—a single binary file that embeds all your .beam files, optionally Elixir .beam files as well, and start-up code. An escript file is thus a fully compiled, cross-platform script that requires only the presence of Erlang on the

running machine. For more details, refer to the `mix escript.build` documentation (http://mng.bz/r1pC).

### 13.1.4. Compiling for production

As mentioned in chapter 10, there is a construct called the *mix environment*—a compile-time identifier that allows you to conditionally define code. The default mix environment is `:dev`, indicating that you're dealing with development. In contrast, when you run tests with `mix test`, the code is compiled in the `:test` environment.

You can use the mix environment to conditionally include code for development or test-time convenience. For example, you can rely on the existence of the `Mix.env` variable to define different versions of a function. Here's a simple sketch:

```
defmodule Todo.Database do
  case Mix.env do
    :dev ->
      def store(key, data) do ... end

    :test ->
      def store(key, data) do ... end

    _ ->
      def store(key, data) do ... end
  end
end
```

Notice how you branch on `Mix.env` on the module level, outside of any functions. This is a compile-time construct, and this code runs during compilation. The final definition of `store/2` will depend on the mix environment you're using to compile the code. In the `:dev` environment, you might run additional logging and benchmarking, whereas in the `:test` environment, you might use an alternative database and perhaps in-memory storage, such as a public ETS table.

It's important to understand that `Mix.env` has meaning only during compilation. You should never rely on it at runtime.

In any case, your code may contain such conditional definitions; even

if it doesn't, a library you depend on may use it. Therefore, you should assume that your project isn't completely optimized when compiled in the `:dev` environment. When running in production, you usually want to use another mix environment, and the prevalent convention is `:prod`.

To start your system in production, you can set the `MIX_ENV` OS environment variable to the corresponding value:

```
$ MIX_ENV=prod elixir -S mix run --no-halt
```

This causes recompilation of the code and all dependencies. All .beam files are stored in the _build/prod folder, and `mix` ensures that the BEAM instance loads files from this folder.

#### Protocol consolidation

Protocols, described in chapter 4, are also important. To refresh your memory, protocols are Elixir's way of implementing polymorphism. For example, you can iterate over all sorts of data structures, such as lists, maps, and streams, using the single `Enum.each/2` function. This function can iterate any structure that implements the `Enumerable` protocol. Internally, `Enum.each/2` makes a polymorphic dispatch to the `Enumerable` protocol, and this dispatch is resolved at runtime. I won't get into the details, but you should be aware that dispatch resolving is by default not as efficient as it could be, mostly in order to support development-time convenience.

To make the protocol dispatch as efficient as possible, you need to *consolidate* protocols. Consolidation analyzes the current state of the project and generates the most efficient dispatching code for each protocol used in the project (and its dependencies). Performing a consolidation is as simple as running `mix compile.protocols`. This is usually needed only when preparing to run in production, so you can also use the `:prod` mix environment:

```
$ MIX_ENV=prod mix compile.protocols
...
Consolidated protocols written to _build/prod/consolidated
```

As the result of the consolidation, you have optimized .beam files in

the _build/prod/consolidated folder. You now need to instruct Elixir to use this folder when looking for binaries:

```
$ MIX_ENV=prod elixir -pa _build/prod/consolidated -S mix run --no-
halt
```

And that's all it takes to optimize protocol dispatch.

**Tip**

It should be obvious from the discussion that the default compile code (in `:dev` mode) isn't as optimal as it could be. This allows for better development convenience, but it makes the code perform less efficiently. When you decide to measure how your system behaves under a heavier load, you should always consolidate protocols and compile everything in the `:prod` environment. Measuring a default `:dev` and nonconsolidated code may give you false indications about bottlenecks, and you may spend energy and time optimizing code that isn't problematic when it's consolidated and compiled in the `:prod` environment.

At this point, you're done with the basics of starting the system with `mix` and `elixir`. This process was mostly simple, and it fits nicely into your development flow.

There are some serious downsides, though. First, to start the project with `mix`, you need to compile it, which means the system source code must reside on the host machine. You need to fetch all dependencies and compile them as well. Consequently, you'll need to install all tools required for compilation on the target host machine. This includes Erlang and Elixir, `hex` and possibly `rebar`, and any other third-party tools that you integrate in your `mix` workflow.

For example, if you're developing a web server, you might use a CoffeeScript compiler to generate the client-side JavaScript code. If you're building your code on the host machine, you'll need to have the CoffeeScript compiler installed there as well.

This means you'll need to pollute the target host machine with compile-time tools. Moreover, if you're running multiple systems on

the same machine, it can become increasingly difficult to reconcile different versions of support tools that are needed for different systems. Luckily, there is a way out, in the form of OTP releases.

## 13.2. OTP releases

An *OTP release* is a standalone, compiled, runnable system that consists of the minimum set of OTP applications needed by the system. An OTP release can optionally include the minimum set of Erlang runtime binaries, which makes the release completely self-sufficient. A release doesn't contain artifacts, such as source code, documentation files, or tests.

This approach provides all sort of benefits. First, you can build the system on your development machine or the build server and ship only binary artifacts. Furthermore, the host machine doesn't need to have any tools installed. If you embed the minimum Erlang runtime into the release, you don't even need Elixir and Erlang installed on the production server. Whatever is required to run the system will be the part of your release package. Finally, releases pave the way for systematic online system upgrades (and downgrades), known in Erlang as *release handling*.

Conceptually, releases seem simple. You need to compile your main OTP application and all of its dependencies and then include all the binaries in the release, together with the Erlang runtime. But the out-of-the-box way to build releases is riddled with mechanical, tedious, error-prone details. A popular approach to building releases in Erlang community is to use a third-party tool such as `rebar` (https://github.com/basho/rebar) or `relx` (https://github.com/erlware/relx), which simplifies the otherwise complex task of building releases.

Elixir version 1.0 doesn't offer special out-of-the-box support for releases. This was a conscious decision, made primarily to reduce scope. But the `exrm` third-party library (https://github.com/bitwalker/exrm) significantly simplifies the release-building process. There have been hints that `exrm` might eventually be integrated into Elixir; by the time this book reaches you, you can verify whether this is already

the case. As it is, `exrm` is currently the best option, and you'll use it to create the release of this book's example to-do system.

### 13.2.1. Building a release with exrm

As you'll see, building a fully standalone release with `exrm` is extremely simple. Including `exrm` as your dependency generates another `mix` task—`mix release`—that you can use to build the release.

#### exrm and Windows support

Unfortunately, at the time of writing, `exrm` doesn't support building releases on the Windows operating system. There are plans to address this issue (and this may happen by the time you read this text), but currently `exrm` isn't suitable for building Windows-based releases.

If you're using Windows, and `exrm` doesn't work for you, you'll need to research other options. A fairly simple (although less abundant in features) way of building releases is to use Erlang's `:systools` module. You can find some initial help in the official Erlang documentation at www.erlang.org/doc/design_principles/release_structure.html.

Let's build your release. First, you need to add `exrm` as a dependency, as shown in the following listing.

**Listing 13.1. Adding `exrm` as a dependency (todo_release/mix.exs)**

```
defmodule Todo.Mixfile do
  ...
  defp deps do
    [
      ...
      {:exrm, "0.14.11"}
    ]
  end
end
```

Notice that you include `exrm` as a compile-time dependency, but you don't list it as a runtime dependency. The sole purpose of `exrm` is to

give you a compile-time `mix release` task. Once you build your release, you won't need `exrm` anymore, and therefore the runtime system doesn't need to list it as a dependency OTP application.

Once you add this dependency, building a release becomes trivial:

```
$ mix deps.get
$ MIX_ENV=prod mix compile --no-debug-info
$ MIX_ENV=prod mix release
...
==> The release for todo-0.0.1 is ready!
```

You may see some warnings in the process, but those can usually be ignored if the final message indicates success.

Notice that while compiling, you're passing a `--no-debug-info` flag. This parameter instructs the Elixir compiler to remove debug information from generated .beam files. As a result, compiled .beam files don't contain debugging-related metadata, which is usually needed only for tools like the debugger or cover analysis. Removing this information makes the files smaller and also prevents the possibility of anyone obtaining the source code (which is possible if debug information is included in the compiled binaries). Also note that `exrm` performs protocol consolidation automatically, so you don't need to worry about this step.

After `mix release` is done, your release resides in the rel/todo subfolder. We'll discuss the release's contents a bit later; first, let's see how you can use it.

### 13.2.2. Using a release

The main tool used to interact with a release is the shell script that resides in rel/todo/bin/todo. You can use it to perform all kinds of tasks, such as these:

- Starting the system and `iex` shell in the foreground
- Starting the system as a background process
- Stopping the running system

- Attaching a remote shell to the running system

The simplest way to verify that the release works is to start the system in the foreground together with the `iex` shell:

```
$ rel/todo/bin/todo console

Starting database worker 1
Starting database worker 2
Starting database worker 3

iex(todo@127.0.0.1)1>
```

Notice from the `iex` shell prompt that the release is automatically running as the `todo` node. By default, `exrm` uses the application name as the node name.

I want to stress that the release is no longer dependent on your system's Erlang and Elixir. It's fully standalone: you can copy the contents of the rel subfolder to another machine where Elixir and Erlang aren't installed, and it will still work. Of course, because the release contains Erlang runtime binaries, the target machine has to be powered by the same OS and architecture.

To start the system as a background process, you can use the `start` argument:

```
$ rel/todo/bin/todo start
```

This isn't the same as a detached process, mentioned earlier. Instead, the system is started via the `run_erl` tool (www.erlang.org/doc/man/run_erl.html). This tool redirects standard output to a log file residing in the rel/todo/log folder, which allows you to analyze your system's console output.

You can then start a remote shell to the node:

```
$ rel/todo/bin/todo remote_console

iex(todo@127.0.0.1)1>
```

Pressing Ctrl-C twice to exit the shell stops the remote shell without disturbing the `todo` node.

It's also possible to *attach* directly to the shell of the running process. On the surface, this looks like the remote shell, but it offers an important benefit: it captures the standard output of the running node. Whatever the running node prints—for example, via `IO.puts`—is seen in the attached process (which isn't the case for the remote shell). To attach to the shell, you use the `attach` argument:

```
$ rel/todo/bin/todo attach
Attaching to /tmp/erl_pipes/todo/erlang.pipe.2 (^D to exit)

iex(todo@127.0.0.1)1>
```

Be careful when attaching to the shell. Unlike a remote shell, an attached shell runs in the context of the running node. You're merely attached to the running node via an OS pipe. The consequence is that stopping the shell by pressing Ctrl-C twice will terminate the running node. As hinted at the prompt, you should press Ctrl-D to exit (detach) from the running node.

If the system is running as a background process, and you want to stop it, you can use the `stop` argument:

```
$ rel/todo/bin/todo stop
```

### 13.2.3. Release contents

Let's spend some time discussing the structure of your release. A fully standalone release consists of the following:

- Compiled OTP applications needed to run your system
- A file containing arguments that will be passed to the virtual machine
- A boot script describing which OTP applications need to be started
- A configuration file containing environment variables for OTP applications
- A helper shell script to start, stop, and interact with the system

- Erlang runtime binaries

In your case, all this resides somewhere in the rel/todo folder. Let's take a closer look at some important parts of the release.

#### Compiled applications

Compiled versions of all required applications reside in the rel/todo/lib folder:

```
$ ls rel/todo/lib

compiler-5.0
cowboy-1.0.0
cowlib-1.0.1
crypto-3.3
elixir-1.0.0
gproc-0.3.1
iex-1.0.0
kernel-3.0
logger-1.0.0
plug-0.10.0
ranch-1.0.0
sasl-2.4
stdlib-2.0
syntax_tools-1.6.14
todo-0.0.1
```

This list includes all of your runtime dependencies, both direct (specified in mix.exs) and indirect (dependencies of dependencies). In addition, some OTP applications, such as `kernel`, `stdlib`, and `elixir`, are automatically included in the release. These are core OTP applications needed by any Elixir-based system. Finally, the `iex` application is also included, which makes it possible to run the remote `iex` shell.

In each of these folders is an ebin subfolder where the compiled binaries reside together with the .app file. Each OTP application folder may also contain the priv folder with additional application-specific files.

#### Tip

If you need to include additional files in the release, the best way to do it is to create a priv folder under your project root. This folder, if it exists, automatically appears in the release under the application

folder. When you need to access a file from the priv folder, you can invoke `Application.app_dir (:an_app_name, "priv")` to find the folder's absolute path.

Bundling all required OTP applications makes the release standalone. Because the system includes all required binaries (including the Elixir and Erlang standard libraries), you don't require anything to exist on the target host machine.

You can prove this by looking at the load paths:

```
$ rel/todo/bin/todo console
iex(todo@127.0.0.1)1> :code.get_path      <-------- Retrieves a list of load paths
['ch12/todo_release/rel/todo/lib/consolidated',
 '.',
 'ch12/todo_release/rel/todo/lib/kernel-3.0/ebin',
 'ch12/todo_release/rel/todo/lib/stdlib-2.0/ebin',
 'ch12/todo_release/rel/todo/lib/compiler-5.0/ebin',
 'ch12/todo_release/rel/todo/lib/syntax_tools-1.6.14/ebin',
 'ch12/todo_release/rel/todo/lib/ranch-1.0.0/ebin',
 'ch12/todo_release/rel/todo/lib/crypto-3.3/ebin',
 'ch12/todo_release/rel/todo/lib/gproc-0.3.1/ebin',
 'ch12/todo_release/rel/todo/lib/cowlib-1.0.1/ebin',
 'ch12/todo_release/rel/todo/lib/elixir-1.0.0/ebin',
 'ch12/todo_release/rel/todo/lib/logger-1.0.0/ebin',
 'ch12/todo_release/rel/todo/lib/cowboy-1.0.0/ebin',
 'ch12/todo_release/rel/todo/lib/plug-0.10.0/ebin',
 'ch12/todo_release/rel/todo/lib/todo-0.0.1/ebin',
 'ch12/todo_release/rel/todo/lib/iex-1.0.0/ebin',
 'ch12/todo_release/rel/todo/lib/sasl-2.4/ebin']
```

Notice how all the load paths point to the release folder. In contrast, when you start a plain `iex -S mix` and run `:code.get_path`, you see a much longer list of load paths, some pointing to the build folder and others pointing to the system Elixir and Erlang installation paths.

This should convince you that your release is self contained. The runtime will only look for modules in the release folder.

For this to work, you need to make sure you specify runtime dependencies in mix.exs. This is used to build a complete list of OTP applications that are needed to run the system. If you fail to list a

dependency application that is needed at runtime, your release will build successfully, but either it will fail to start or you'll experience various runtime errors.

#### Erlang runtime

As mentioned, the minimum Erlang binaries are included in the release. They reside in rel/todo/erts-X.Y, where X.Y corresponds to the runtime version number (which isn't related to the Erlang version number).

The fact that the Erlang runtime is included makes the release completely standalone. Moreover, it allows you to run different systems on different Erlang versions.

But embedding the Erlang runtime also ties the release to the particular OS version and architecture. If this doesn't suit your needs, you can remove the runtime from the release. Currently, the simplest way to do this is to create a file called todo/rel/relx.config and place the `{include_erts, false}.` option in it (note the ending dot character). Then you can regenerate the release, which now won't contain the Erlang runtime. Nothing else needs to be changed, and you can use the release exactly as before. The release will rely on the existence of the appropriate Erlang runtime, which must be available in the path.

#### Configurations

Configuration files reside in the rel/todo/releases/0.0.1 folder, with 0.0.1 corresponding to the version of your `todo` application (as provided in mix.exs). The two most relevant files in this folder are vm.args and sys.config.

The vm.args file can be used to provide flags to the Erlang runtime, such as the aforementioned `+P` flag that sets the maximum number of running processes. Some basic defaults are generated for you by `exrm`. For example, the node name and magical cookie are specified here. You can of course provide your own vm.args file as an input to `exrm`. Refer to the official project repository (https://github.com/bitwalker/exrm) for details.

sys.config is an Erlang file that contains OTP environment variables, as specified in your mix.exs and config.exs files. For example, in chapter 10, you made the HTTP listening port configurable via an environment variable, and in config.exs you've set it to the value 5454. This setting will propagate to the sys.config file. When you start the system, sys.config will be consulted, and the corresponding environment variables will be set prior to starting the OTP applications.

Consequently, sys.config can be used to tweak settings without the need to rebuild the release. For example, if you want to change the listening port, you can edit sys.config on the target machine and restart the system.

#### Compressed release package

If you take a closer look at the contents of the rel/todo folder, you'll notice a compressed tarball named todo-0.0.1.tar.gz, which is essentially a compressed version of the entire release. When you want to deploy to the target machine, you can copy this file, unpack it, and start the system by running `bin/todo start`.

This file isn't just a convenience for faster upload. It plays an important role in the live upgrade process. When you make changes to your code, you can increase the version number in mix.exs and rebuild the release. The resulting release package will have a different name (such as todo-0.0.2.tar.gz). You can then upload this file to the releases subfolder of your running system and invoke `bin/test upgrade "0.0.2"`, which will make your system upgrade on the fly, without restarting. See the `exrm` documentation for details.

### Live upgrades can be tricky

In its basic version, a live upgrade can look deceptively simple, especially with the help of the `exrm` tool. But in a more complex project, live upgrading can become much trickier. Keep in mind that a BEAM system runs many stateful server processes and may maintain state in ETS tables. A more complex upgrade may require implementing the `code_changegen_server` callback, where you need to migrate the old process state to the new one. Moreover, you'll sometimes need to reconfigure your supervision tree or even stop or

restart some OTP applications. Doing this properly requires careful changes and exhaustive testing. You should therefore first consider whether you can tolerate the short downtime caused by plain BEAM process restarts.

If you're running your system as a cluster, you can also migrate nodes one by one, thus keeping the entire system running. Resort to live upgrades only when you establish that simple restarts aren't good enough for your particular use case. For details, see the `exrm` documentation and Erlang documentation (www.erlang.org/doc/design_principles/release_handling.html).

#### Fine-grained release assembly

Tools such as `exrm` assemble the entire release in a single pass on the build machine. This should serve most purposes, but sometimes it may not fit your particular needs. For example, you may want to deploy your system on a large number of different machines. In such cases, embedding the Erlang runtime won't work, because it's tied to the particular OS and architecture. It's more likely you'll want to assemble just the .beam and configuration files on the build server, finishing the release assembly process on each target machine. It's even possible to have your node load the code from a remote machine, which means your release doesn't need to contain application binaries.

Obviously, there are all sorts of options; if you have special requirements, you'll need to roll up your sleeves and assemble some parts of the release on your own. To do this, you'll need to learn about releases in more detail. Erlang's documentation (www.erlang.org/doc/design_principles/release_structure.html) is a good first source. In addition, you may want to research the documentation for `:systools` (www.erlang.org/doc/man/systools.html) and `:reltool` (www.erlang.org/doc/man/reltool.html).

This concludes the topic of releases. Once you have your system up and running, it's useful to see how you can analyze its behavior.

## 13.3. Analyzing system behavior

Even after the system is built and placed in production, your work isn't done. Things will occasionally go wrong, and you'll experience errors. The code also may not be properly optimized, and you may end up consuming too many resources. If you manage to properly implement a fault-tolerant system, it may recover and cope with the errors and increased load. Regardless, you'll still need to get to the bottom of any issues and fix them.

Given that your system is highly concurrent and distributed, it may not be obvious how you can discover and understand the issues that arise. Proper treatment of this topic could easily fill a separate book—and an excellent free book is available, called *Stuff Goes Bad: Erlang in Anger*, by Fred Hébert (http://www.erlang-in-anger.com). This chapter provides a basic introduction to some standard techniques of analyzing complex BEAM systems. But if you plan to run Elixir/Erlang code in production, you should at some point study the topic in more detail, and *Erlang in Anger* is a great place to start.

### 13.3.1. Debugging

Although it's not strictly related to the running system, debugging deserves a brief mention. It may come as a surprise that standard step-by-step debugging isn't a frequently used approach in Erlang (which ships with a GUI-based debugger; see www.erlang.org/doc/apps/debugger/debugger_chapter.html). The reason is that it's impossible to do a classical debugging of a highly concurrent system, where many things happen simultaneously. Imagine that you set a breakpoint in some process.

What should happen to other processes when the breakpoint is encountered? Should they continue running, or should they pause as well? Once you step over a line, should all other processes move forward by a single step? How should timeouts be handled? What happens if you're debugging a distributed system? As you can see, there are many problems with classical debugging, due to the highly concurrent and distributed nature of BEAM-powered systems.

So instead of relying on a debugger, you should adapt to more appropriate strategies. The key to understanding a highly concurrent

system lies in logging and tracing. Once something goes wrong, you'll want to have as much information as possible, which will allow you to find the cause of the problems.

The nice thing is that some logging is available out of the box in the form of Elixir's `logger` application (http://elixir-lang.org/docs/stable/logger/). In particular, whenever an OTP-compliant process crashes (such as `gen_server`), an error is printed, together with a stack trace. The stack trace also contains file and line information, so this should serve as a good starting point to investigate the error.

Sometimes the failure reason may not be obvious from the stack trace, and you need more data. At development time, a primitive helper tool for this purpose is `IO.inspect`. Remember that `IO.inspect` takes an expression, prints its result, and returns it. This means you can surround any part of the code with `IO.inspect` (or pipe into it via `|>`) without affecting the behavior of the program. This is a simple technique that can help you quickly determine the cause of the problem, and I use it frequently when a new piece of code goes wrong. Placing `IO.inspect` to see how values were propagated to the failing location often helps me discover errors. Once I'm done fixing the problem, I remove the `IO.inspect` calls.

Another useful feature is *pry*, which allows you to temporarily stop execution in the `iex` shell and inspect the state of the system, such as variables that are in scope. For detailed instructions, refer to the `IEx.pry/1` documentation (http://elixir-lang.org/docs/stable/iex/IEx.html#pry/1).

Of course, it goes without saying that automated tests can be of significant assistance. Testing individual parts in isolation can help you quickly discover and fix errors.

It's also worth mentioning a couple of useful benchmarking and profiling tools. The most primitive one comes in the form of the `:timer.tc/1` function (www.erlang.org/doc/man/timer.html#tc-1), which takes a lambda, runs it, and returns its result together with the running time (in microseconds).

A somewhat more structured helper library for benchmarking is `Benchfella` (https://github.com/alco/benchfella), which integrates nicely with the `mix` tool and provides a framework for generating automated benchmarking tests.

In addition, a few profiling tools are shipped with Erlang/OTP: `eprof`, `fprof`, and `cprof`. I won't explain them in detail, so when you decide to profile, it's best to start reading the official Erlang documentation (www.erlang.org/doc/efficiency_guide/profiling.html). If you want to analyze the system's concurrent behavior, you may find the `percept` tool (www.erlang.org/doc/apps/percept/percept_ug.html) useful as well.

### 13.3.2. Logging

Once you're in production, you shouldn't rely on `IO.inspect` calls anymore. Instead, it's better to log various pieces of information that may help you understand what went wrong. For this purpose, you can rely on Elixir's `logger` application. When you generate your `mix` project, this dependency will be included automatically, and you're encouraged to use `logger` to log various events. As already mentioned, `logger` automatically catches various BEAM reports, such as crash errors that happen in processes.

Logging information by default goes to the console. If you start your system as a release, the standard output will be forwarded to the log folder under the root folder of your release, and you'll be able to later find and analyze those errors.

Of course, you can write a custom `logger` backend—for example, the one that writes to `syslog` or sends log reports to a different machine. See the `logger` documentation for more details (http://elixir-lang.org/docs/stable/logger/).

### 13.3.3. Interacting with the system

A huge benefit of the Erlang runtime is that you can connect to the

running node and interact with it in various ways. You can send messages to processes, and stop or restart different processes (including supervisors) or OTP applications. It's even possible to force the VM to reload the code for a module.

On top of all this, all sorts of built-in functions allow you to gather data about the system and individual processes. For example, you can start a remote shell and use functions such as `:erlang.system_info/1` and `:erlang.memory/0` to get information about the runtime.

You can also get a list of all processes using `Process.list/0` and then query each process in detail with `Process.info/1`, which returns information such as memory usage and total number of instructions (known in Erlang as *reductions*) the process has executed. Such services make way for tools that can connect to the running system and present BEAM system information in a GUI.

One example is the `observer` application (www.erlang.org/doc/man/observer.html). Being GUI based, `observer` works only when there's a windowing system in the host OS. On the production server, this usually isn't the case. But you can start the observer locally and have it gather data from a remote node.

Let's see this in action. You'll start your system as a background service and then start another node on which you'll run the `observer` application. The `observer` application will connect to the remote node, collect data from it, and present it in the GUI.

The production system doesn't need to run the `observer` application, but it needs to contain a module that gathers data for the remote `observer` application. This module is part of the `runtime_tools` application that you need to include as a runtime dependency, as shown in the following listing.

**Listing 13.2. Adding `runtime_tools` as a dependency (todo_runtime/mix.exs)**

```
defmodule Todo.Mixfile do
  ...
```

```
  def application do
    [
      applications: [:runtime_tools, ...],
      ...
    ]

  ...
end
```

Notice that you don't include `runtime_tools` as a compile-time dependency. The reason is that a compiled `runtime_tools` already exists on the disk as the part of the standard Erlang/OTP distribution, so you don't need to fetch it from anywhere. You still need to specify it as a runtime dependency so that binaries of this OTP application are included in the release.

Once you've done this, you need to rebuild the release (make sure to stop any previous to-do system that was running) and run it in background:

```
$ MIX_ENV=prod mix compile
$ MIX_ENV=prod mix release
$ rel/todo/bin/todo start
```

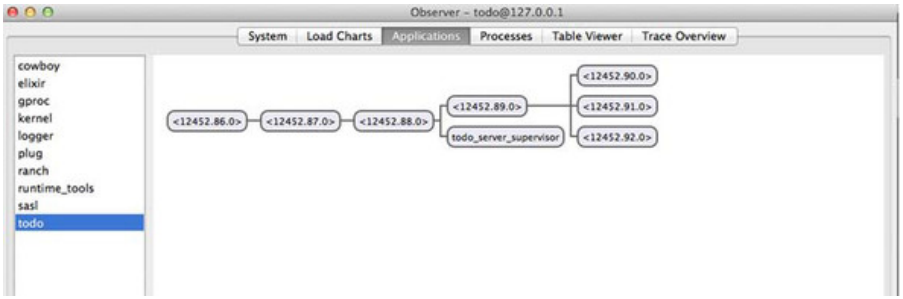
Finally, start the interactive shell as a named node, and start the `observer` application:

```
$ iex --hidden --name observer@127.0.0.1 --cookie todo

iex(observer@127.0.0.1)1> :observer.start
```

Note how you explicitly set the node's cookie to match the one used in the running system. Also, just as with the earlier `remsh` example, you start the node as hidden.

Once the observer is started, you need to select Nodes > Connect Node from the menu and provide `todo@127.0.0.1` as the node you want to connect to. At this point, `observer` is monitoring the todo node. Clicking on the Processes tab provides a view of the processes. You can do many other nice things in `observer`, such as analyze the contents of ETS tables and inspect your application's supervision tree, as shown in figure 13.1.

**Figure 13.1. Inspecting the supervision tree of the `todo` application**

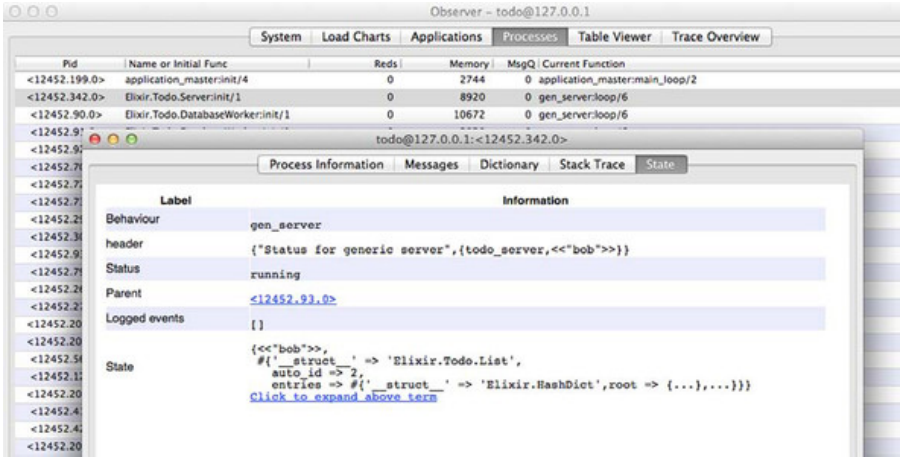



You can even analyze the state of an individual process. To do this, try the following:

1. Open the Processes tab.
2. Sort by the Name or Initial Func column.
3. In the OS terminal, fire up an HTTP request for Bob's list: `curl "http://localhost:5454/entries?list=bob&date=20131219"`.
4. Back in `observer`, find the row where the Name or Initial Func column has the value Elixir.Todo.Server:init/1.
5. Double-click the row, and, in the new window, click the State tab.

You should get a result similar to that shown in figure 13.2.

**Figure 13.2. Inspecting a process state in the `observer` application**




The underlying `runtime_tools` application that gathers all this data is implemented in pure Erlang. It relies on various functions that allow you to introspect the BEAM instance and individual processes.

Sometimes you won't be able to use `observer`. For example, I had a case where I could only use SSH to connect to the remote server, so I couldn't set up a remote `iex` session to the running system.

In such cases, the only option is to start an SSH session to the server and, from the server, start the `iex` session to the running BEAM. But now you can't use a graphical interface (because there's probably no windowing system on the target server). At this point, you need to resort to text-based tools. A simple tool that ships with the standard Erlang/OTP distribution is `etop` (www.erlang.org/doc/apps/observer/etop_ug.html)—a kind of `top` equivalent for BEAM. Personally, I find `etop` too rudimentary, and I tend to use a third-party library called `entop` (https://github.com/mazenharake/entop), which has a few more features.

If you want to collect statistics and ship them to another component (such as Graphite), you should look into libraries such as `Folsom` (https://github.com/boundary/folsom) and Exometer (https://github.com/Feuerlabs/exometer). Finally, the Recon library (https://github.com/ferd/recon) offers many useful functions for inspecting a running BEAM system.

### 13.3.4. Tracing

It's also possible to be able to turn on traces related to processes and function calls, relying on services from the `:sys` (www.erlang.org/doc/man/sys.html) and `:dbg` modules. The `:sys` module allows you to trace OTP-compliant processes (for example, `gen_server`). Tracing is done on the standard output, so you need to attach to the system (as opposed to establishing a remote shell). Then, you can turn tracing for a particular process with the help of `:sys.trace/2`:

```
$ rel/todo/bin/todo attach
iex(todo@127.0.0.1)1> Todo.Cache.server_process("bob") |>
                      :sys.trace(true)
```

This turns on console tracing. Information about process-related events, such as received requests, will be printed to the standard output.

Now, let's issue an HTTP request for Bob's list:

```
$ curl "http://localhost:5454/entries?list=bob&date=20131219"
```

Back in the attached shell, you should see something like this:

```
*DBG* {todo_server,<<"bob">>} got call {entries,{2013,12,19}}
      from <0.322.0>

*DBG* {todo_server,<<"bob">>} sent [] to <0.322.0>,
      new state {<<98,111,98>>,#{'__struct__'=>'Elixir.Todo.List',
        auto_id=>1,entries=>#{'__struct__'=>'Elixir.HashDict',
        root=>{[],[],[],[],[],[],[],[]},size=>0}}
      }
```

The output may seem a bit cryptic, but if you look carefully, you can see two trace entries: one for a received call request and another for the response you sent. You can also see the full state of the server process. Keep in mind that all terms are printed in Erlang syntax.

Tracing is a powerful tool, because it allow you to analyze the behavior of the running system. But you should be careful, because excessive tracing may hurt the system's performance. If the server process you're tracing is heavily loaded or has a huge state, BEAM will

spend a lot of time doing tracing I/O, which may slow down the entire system.

In any case, once you've gathered some knowledge about the process, you should stop tracing it:

```
iex(todo@127.0.0.1)1> Todo.Cache.server_process("bob") |>
                      :sys.trace(false)
```

Other useful services from `:sys` allow you to get the OTP process state (`:sys.get_ state/1`) and even change it (`:sys.replace_state/2`). Those functions are meant to be used purely for debugging or hacky manual fixes, and you shouldn't invoke them from your code.

Another useful tracing tool comes with the `:erlang.trace/3` function (http://erlang.org/doc/man/erlang.html#trace-3), which allows you to subscribe to events in the system such as message passing or function calls. In addition to this function, an entire module called `:dbg` (http://erlang.org/doc/man/dbg.html) is available, which simplifies tracing.

You can run `:dbg` directly on the attached console, but it's also possible to start another node and make it trace the main system. This is the route you'll take in the next example. Assuming your to-do node is running, start another node:

```
$ iex --name tracer@127.0.0.1 --cookie todo --hidden
```

Now, on the `tracer` node, start tracing the main `todo` node, and specify that you're interested in all calls to functions from the `Todo.Server` module:

Starts the tracer process
iex(tracer@127.0.0.1)1> :dbg.tracer
iex(tracer@127.0.0.1)2> :dbg.n(:'todo@127.0.0.1')
Subscribes only to events from the todo node
Subscribes to function calls in all processes
iex(tracer@127.0.0.1)3> :dbg.p(:all, [:call])
iex(tracer@127.0.0.1)4> :dbg.tp(Todo.Server, [])
Sets the trace pattern to all functions from the Todo.Server process

With traces set up, you can make an HTTP request to retrieve Bob's

entries. In the shell of the `tracer` node, you should see something like the following:

```
(<8936.344.0>) call 'Elixir.Todo.Server':whereis(<<"bob">>)

(<8936.344.0>) call 'Elixir.Todo.Server':entries(<8936.336.0>,
                      {2013,12,19})

(<8936.336.0>) call 'Elixir.Todo.Server':handle_call(
                      {entries,{2013,12,19}},
                      {<8936.344.0>,#Ref<8936.0.0.686>},
                      {<<"bob">>, #{'__struct__' =>
'Elixir.Todo.List',
                         auto_id => 1,
                         entries => #{'__struct__' =>
'Elixir.HashDict',
                         root => {[],[],[],[],[],[],[],[]},size =>
0}}
                      })
```

Again, be careful about tracing production, because huge number of traces may flood the system. Once you're finished tracing, invoke `:dbg.stop_clear/0` to stop all traces.

This was admittedly a brief demo; `:dbg` has many more options. If you decide to do some tracing, you should look at the `:dbg` documentation. There is also a more user-friendly Elixir wrapper library available: the third-party `dbg` library (https://github.com/fishcakez/dbg).

## 13.4. Summary

In this chapter, you've learned how to start the entire system and bundle it into a standalone OTP release. I also discussed some techniques for interacting with a live system. Several points are worth repeating:

- To start a system, all code must be compiled. Then you must start a BEAM instance with properly set up load paths. Finally, you need to start all OTP applications.
- The simplest way to do this is to rely on Elixir tools such as `iex` and `mix`.
- An OTP release is a standalone system consisting only of

runtime artifacts—compiled OTP applications and (optionally) the Erlang runtime system.

- When you're building for production, use the `prodmix` environment. If you're running the system with `mix`, remember to consolidate protocols.
- Once the system is running, you can connect to it via a remote shell or attach to its console. Then you can interact with the system in various ways and find detailed information about the VM and individual processes.

We're finished exploring Elixir, Erlang, and OTP. This book covered the primary aspects of the Elixir language, basic functional programming idioms, the Erlang concurrency model, and the most frequently used OTP behaviours (`gen_server`, `supervisor`, and `application`). In my experience, these are the most frequently needed building blocks of Elixir and Erlang systems.

Of course, many topics have been left untreated, so your journey doesn't stop here. This book was never meant to be an exhaustive reference to Elixir, Erlang, and OTP. You'll probably want to look for other knowledge resources, such as other books, blogs, and podcasts. A good starting place to look for further material is the Elixir Wiki (https://github.com/elixir-lang/elixir/wiki).

# Index

[SYMBOL][A][B][C][D][E][F][G][H][I][K][L][M][N][O][P][Q][R][S][T][U][V][W]

## SYMBOL

^ (pin operator)
; (expression separator)
! (exclamation point), 2nd
!= (weak inequality operator)
!== (strict inequality operator)
? (question mark), 2nd
. (function application)
[ ] operator, 2nd
/ (division operator)
\ (string escaping)
\\ (optional arguments)
& (capture operator), 2nd
# (comment)
<< and >> operators
<> operator
= (match operator), 2nd
== (weak equality operator)
=== (strict equality operator)
|> (pipeline operator), 2nd, 3rd, 4th

## A

abstract syntax tree.
    *See* AST.
Access protocol
Actor model
agents
alias registration
aliases, 2nd, 3rd, 4th
anonymous functions
anonymous variables
application behaviour, 2nd
arguments, function
arity of functions
Armstrong, Joe
AST (abstract syntax tree)
asynchronous requests
attributes, module

## B

background processes
bag type (ETS tables)

BEAM (Bogdan/Björn's Erlang Abstract Machine), 2nd
.beam extension
behaviours, OTP
Benchfella library
binary strings, 2nd
binding variables, 2nd
bitstrings, 2nd
Bogdan/Björn's Erlang Abstract Machine.
    See BEAM.
booleans
bottlenecks
  process as, 2nd
  for processes
    bypassing process
    handling requests concurrently
    limiting concurrency with pooling
bytecode

## C

cache
  concurrency using
  using ETS tables
calls vs. casts
CAP theorem
capture operator ( & ), 2nd
case macro
casts, 2nd, 3rd, 4th, 5th
catch block
character lists
closures
collectables, 2nd
comments
comparison operators
compile time
compiling for production
compound matches
comprehensions
concurrency
  in Erlang
  errors
    linking processes
    monitors
    overview
    trapping exits
  message passing
    collecting query results example
    overview
    receive algorithm
    synchronous sending
  overview, 2nd
  parallelism vs.
  persisting data
    encoding and decoding data
    overview
    reading from database
    storing in database

process bottleneck
bypassing process
handling requests concurrently
limiting concurrency with pooling
processes
creating
overview
runtime
bottlenecks
process mailboxes
scheduler overview
shared nothing concurrency
stateful server processes
complex states
functional programming and
iterations using
maintaining state
mutable state
overview
registered processes
synchronous nature of
to-do lists example
overview
process dependencies and
using cache
using mix tool
concurrency-oriented language
conditionals
branching with multiclause functions
case macro
cond macro
if macro
unless macro
connection pool
consistent hashing
constants
cookies
Cowboy library
cprof tool

## D

dbg library
debugging
decoding data
def/defmodule macros
defexception macro
defprotocol macro
defrecord/defrecordp macros
deployment
deployable systems
OTP releases
building with exrm
compiled applications in
compressed release package in
configuration files in

Erlang binaries in
manual assembly of
overview
using releases
system analysis
debugging
logging
system interaction
tracing
using mix tool
bypassing shell
compiling for production
overview
protocol consolidation
running as background process
running scripts
detached mode
DETS (disk-based ETS)
dialyzer tool
disconnected nodes
division operator ( / )
duplicate_bag type (ETS tables)
dynamic programming languages

## E

ecosystem for Erlang
Ecto library, 2nd
elixir command
elixir-lang mailing list
encoding data
entop library
enumerables, 2nd, 3rd, 4th
EPMD (Erlang Port Mapper Daemon)
eprof tool
--erl option
ERL_MAX_ETS_TABLES environment variable
Erlang Port Mapper Daemon.
See EPMD.
Erlang Term Storage tables.
See ETS tables.
escripts
etop tool
ETS (Erlang Term Storage) tables
basic operations
DETS
key-based lookups
match patterns
match specifications
Mnesia
overview
page cache using
traversing
use cases for
ETS permissions
private permission
protected permission

public permission
.ex extension
ex_doc tool
ExActor library
exclamation point ( ! ), 2nd
exits, 2nd
Exometer library
exporting functions
exrm library
ExUnit

## F

firewalls
Folsom library
fprof tool
functional application ( . )

## G

garbage collection, 2nd, 3rd
gen_event behaviour
gen_fsm behaviour
GenServer module, 2nd
global registration
gproc library, 2nd, 3rd
group leader
guards

## H

h command
hash character ( # )
HashDict
HashSet
heredocs syntax
Hex package manager, 2nd
hidden nodes
hierarchical data
generating IDs
immutable hierarchical updates
iterative updates
updating entries
high availability, 2nd
higher-order functions

## I

iex command
if macro
implementation functions
importing modules
intentional programming
interface functions, 2nd, 3rd, 4th
IO lists

## K

keyword lists

## L

library applications
line breaks
linking processes, 2nd, 3rd
live upgrades, 2nd
local registration
location transparency
logger application
logical operators
loops.
    *See* iterations.

## M

macros, 2nd
mailboxes, process
match operator ( = )
match patterns
match specifications
maximum restart frequency
MFA (module, function, arguments list)
MIX_ENV OS environment variable
Mnesia
modifier functions
module, function, arguments list.
    *See* MFA.
monitors, 2nd
multiclause functions, 2nd
multiclause lambdas

## N

--name option
named_table option
netsplits

defined
detecting
nil
Node module
--no-debug-info option
--no-halt option, 2nd, 3rd
number types

## O

observer tool, 2nd
one_for_all strategy
operators
optional arguments ( \\ )
ordered_set type (ETS tables)
OTP (Open Telecom Platform), 2nd
OTP behaviours
OTP-compliant processes, 2nd
overloading functions

## P

parallelism
parse transforms
partitions
performance
Erlang and
protocols and
web server
persistent state
persisting data
encoding and decoding data
overview
process bottleneck
bypassing process
handling requests concurrently
limiting concurrency with pooling
reading from database
storing in database
Phoenix web framework
pid, 2nd
pin operator ( ^ )
pipeline operator ( |> ), 2nd, 3rd, 4th
Plug library
poolboy
pooling
port identifier
ports
Profiler module
profiling tools
project environment
pry

## Q

question mark ( ? ), 2nd

## R

raising errors
ranges
rebar tool
receive algorithm
Recon library
records
recursion
  iterations with
  tail vs. non-tail
reductions
references
registration
  of attributes
  global
  local
  of processes
relx tool
remote processes
--remsh option
rest_for_one strategy
restart frequency
restart strategies, 2nd
rpc module
run queue
run_erl tool
runner modules

## S

scheduler, BEAM
scripts, running, 2nd
security
semicolon ( ; )
server_process request, 2nd
ServerProcess module, 2nd
server-side systems
set type (ETS tables)
shared nothing concurrency
shell.
    See interactive shell.
short-circuit operators
side effects
simple_one_for_one supervisors
--sname option
spawning lambdas
special forms
split-brain situation

staircasing
stdlib application
streams
strict equality ( === ) operator
strict inequality ( !== ) operator
string escaping ( \ )
String.Chars protocol
strings
  binary
  character lists
structs
  abstractions using
  maps vs.
--sup option
supervision trees
  defined
  organizing
    avoiding process restarting
    OTP-compliant processes
    overview
    shutting down processes
  process discovery
  removing database process
  separating loosely dependent parts
  starting system
  supervising database workers
supervisor behaviour
supervisor child specification
synchronized code

## T

tail function calls, 2nd
tasks
temporary workers
throwing errors
timeouts
tracing
transient workers
trapping exits
truthy values
try block
type specifications

## U

underscore ( _ ), 2nd
unit tests
unless macro, 2nd
use macro

## V

Valim, José
via tuples, 2nd
visibility of functions
visible nodes

## W

weak equality ( == ) operator
weak inequality ( != ) operator

# List of Figures


Chapter 1. First steps

Figure 1.1. Concurrency in the Erlang virtual machine

Figure 1.2. Server-side system

Chapter 2. Building blocks

Figure 2.1. Recursive structure of the list [1, 2, 3, 4]

Figure 2.2. Modifying a tuple creates a shallow copy of it.

Figure 2.3. Rebinding a tuple makes the old data garbage-collectible.

Figure 2.4. Modifying a list

Figure 2.5. Pushing a new element to the top of the list

Chapter 5. Concurrency primitives

Figure 5.1. BEAM as a single OS process, using a few threads to schedule a large number of processes

Figure 5.2. Inter-process communication via messages

Figure 5.3. Synchronous send and receive implemented on top of the asynchronous messages

Figure 5.4. m:n threading: a small number of schedulers running a large number of BEAM processes

Chapter 6. Generic server processes

Figure 6.1. Life cycle of a gen_server-powered process

Chapter 7. Building a concurrent system

Figure 7.1. Cooperation of processes

Figure 7.2. Process dependencies

Figure 7.3. Handling requests concurrently

Figure 7.4. Pooling database operations

Chapter 8. Fault-tolerance basics

Figure 8.1. Isolating errors in the to-do system

Figure 8.2. Example of links with multiple processes

Figure 8.3. Processes in the to-do system

Figure 8.4. Relationships between supervisor and worker processes

Figure 8.5. Linking all processes in the to-do system

Chapter 9. Isolating error effects

Figure 9.1. Process links in the to-do system

Figure 9.2. Separated supervision of the database and the cache

Figure 9.3. Discovering processes through a registry

Figure 9.4. Using the process registry to discover supervised database workers

Figure 9.5. Supervision tree

Figure 9.6. Supervising to-do servers

Figure 9.7. rest_for_one supervision

Chapter 10. Sharing state

Figure 10.1. Managing the cache in the ETS table

Figure 10.2. Finer-grained write synchronization

Figure 10.3. The process registry is a bottleneck.

Chapter 11. Working with components

Figure 11.1. Requests are handled in separate processes

Figure 11.2. Independent simultaneous requests are handled concurrently, whereas requests on the same to-do list are synchronized.

Figure 11.3. Using casts reduces the certainty of your responses.

Figure 11.4. Using calls promotes consistency but reduces the responsiveness of the system.

Figure 11.5. Queuing the request and sending a notification about the status

## Chapter 12. Building a distributed system

Figure 12.1. The to-do system as a cluster

Figure 12.2. Using a to-do cache

Figure 12.3. Replicating the database

Figure 12.4. Connecting to a remote node

## Chapter 13. Running the system

Figure 13.1. Inspecting the supervision tree of the todo application

Figure 13.2. Inspecting a process state in the observer application

# List of Tables

Chapter 1. First steps

Table 1.1. Comparison of technologies used in two real-life web servers

Chapter 2. Building blocks

Table 2.1. Comparison operators

# List of Listings


Chapter 1. First steps

Listing 1.1. Erlang-based server process that adds two numbers

Listing 1.2. Elixir-based server process that adds two numbers

Listing 1.3. Elixir-based server process to add numbers based on a third-party abstraction

Chapter 2. Building blocks

Listing 2.1. Defining a module (geometry.ex)

Listing 2.2. Functions with the same name but different arities (arity_demo.ex)

Listing 2.3. Same-name functions, different arities, default params (arity_calc.ex)

Listing 2.4. Module with a public and a private function (private_fun.ex)

Listing 2.5. Elixir script (script.exs)

Chapter 3. Control flow

Listing 3.1. Pattern-matching function arguments (rect.ex)

Listing 3.2. Multiclause function (geometry.ex)

Listing 3.3. Multiclause function (geometry_invalid_input.ex)

Listing 3.4. Using guards (test_num.ex)

Listing 3.5. Using guards (test_num2.ex)

Listing 3.6. Counting the lines in a file (lines_counter.ex)

Listing 3.7. Printing the first n natural numbers (natural_nums.ex)

Listing 3.8. Calculating the sum of the list (sum_list.ex)

Listing 3.9. Tail-recursive sum of the first n natural numbers

(sum_list_tc.ex)

Chapter 4. Data abstractions

Listing 4.1. Initializing a to-do list (simple_todo.ex)

Listing 4.2. Adding an entry (simple_todo.ex)

Listing 4.3. Querying the to-do list (simple_todo.ex)

Listing 4.4. Implementing the MultiDict abstraction (todo–multi_dict.ex)

Listing 4.5. TodoList relying on a MultiDict (todo_multi_dict.ex)

Listing 4.6. Representing entries with maps (todo_entry_map.ex)

Listing 4.7. Defining a structure (fraction.ex)

Listing 4.8. Instantiating a fraction (fraction.ex)

Listing 4.9. Calculating the fraction value (fraction.ex)

Listing 4.10. Adding two fractions (fraction.ex)

Listing 4.11. TodoList struct (todo_crud.ex)

Listing 4.12. Autogenerating ID values for new entries (todo_crud.ex)

Listing 4.13. Filtering entries for a given date (todo_crud.ex)

Listing 4.14. Updating an entry (todo_crud.ex)

Listing 4.15. Iteratively building the to-do list (todo_builder.ex)

Chapter 5. Concurrency primitives

Listing 5.1. Long-running server process that runs queries (database_server.ex)

Listing 5.2. Database server loop (database_server.ex)

Listing 5.3. Implementation of run_async/2 (database_server.ex)

Listing 5.4. Implementation of get_result/0 (database_server.ex)

Listing 5.5. Initializing the process state (stateful_database_server.ex)

Listing 5.6. Using the connection while querying (stateful_database_server.ex)

Listing 5.7. Concurrent stateful calculator (calculator.ex)

Listing 5.8. Calculator interface functions (calculator.ex)

Listing 5.9. TodoServer modules (todo_server.ex)

Listing 5.10. TodoServer basic structure (todo_server.ex)

Listing 5.11. The add_entry request (todo_server.ex)

Listing 5.12. The entries request (todo_server.ex)

Listing 5.13. Demonstration of a process bottleneck (process_bottleneck.ex)

Chapter 6. Generic server processes

Listing 6.1. Starting the server process (server_process.ex)

Listing 6.2. Handling messages in the server process (server_process.ex)

Listing 6.3. Helper for issuing requests (server_process.ex)

Listing 6.4. Key-value store implementation (server_process.ex)

Listing 6.5. Wrapping ServerProcess function calls (server_process.ex)

Listing 6.6. Including the request type in the message (server_process_cast.ex)

Listing 6.7. Supporting casts in the server process (server_process_cast.ex)

Listing 6.8. Implementing put as a cast (server_process_cast.ex)

Listing 6.9. Implementing gen_server callbacks (key_value_gen_server.ex)

Listing 6.10. Adding interface functions (key_value_gen_server.ex)

Chapter 7. Building a concurrent system

Listing 7.1. Cache initialization (todo_cache/lib/todo/cache.ex)

Listing 7.2. Handling the server_process request (todo_cache/lib/todo/cache.ex)

Listing 7.3. Interface functions (todo_cache/lib/todo/cache.ex)

Listing 7.4. Database process (persistable_todo_cache/lib/todo/database.ex)

Listing 7.5. Starting the database (persistable_todo_cache/lib/todo/cache.ex)

Listing 7.6. Persisting the data (persistable_todo_cache/lib/todo/server.ex)

Listing 7.7. Reading data (persistable_todo_cache/lib/todo/server.ex)

Chapter 8. Fault-tolerance basics

Listing 8.1. Changes in the to-do cache (supervised_todo_cache/lib/todo/cache.ex)

Listing 8.2. To-do cache supervisor (supervised_todo_cache/lib/todo/supervisor.ex)

Listing 8.3. Starting the supervisor (supervised_todo_cache/lib/todo/supervisor.ex)

Chapter 9. Isolating error effects

Listing 9.1. Supervising database server (supervise_database/lib/todo/supervisor.ex)

Listing 9.2. Registering a process (pool_supervision/lib/todo/process_registry.ex)

Listing 9.3. Registering workers (pool_supervision/lib/todo/database_worker.ex)

Listing 9.4. Worker functions (pool_supervision/lib/todo/database_worker.ex)

Listing 9.5. Pool supervisor (pool_supervision/lib/todo/pool_supervisor.ex)

Listing 9.6. Starting the pool (pool_supervision/lib/todo/database.ex)

Listing 9.7. Database requests (pool_supervision/lib/todo/database.ex)

Listing 9.8. To-do server supervisor (dynamic_workers/lib/todo/server_supervisor.ex)

Listing 9.9. Todo.Supervisor (dynamic_workers/lib/todo/supervisor.ex)

Chapter 10. Sharing state

Listing 10.1. Long processing of an index request (profile_cache/lib/web_server.ex)

Listing 10.2. Initializing an ETS powered cache (profile_cache/lib/ets_page_cache.ex)

Listing 10.3. Reading from the cache (profile_cache/lib/ets_page_cache.ex)

Listing 10.4. Writing into the cache (profile_cache/lib/ets_page_cache.ex)

Chapter 11. Working with components

Listing 11.1. Specifying application parameters (todo_app/mix.exs)

Listing 11.2. Implementing the application callback (todo_app/lib/todo/application.ex)

Listing 11.3. Reusing gproc in your project (gproc_todo_app/mix.exs)

Listing 11.4. Depending on the gproc application at runtime (gproc_todo_app/mix.exs)

Listing 11.5. Simplified top-level supervisor (gproc_todo_app/lib/todo/supervisor.ex)

Listing 11.6. External dependencies for the web server (todo_web/mix.exs)

Listing 11.7. Starting the server (todo_web/lib/todo/web.ex)

Listing 11.8. Starting the HTTP server (todo_web/lib/todo/application.ex)

Listing 11.9. Setting up a route for add_entry (todo_web/lib/todo/web.ex)

Listing 11.10. Implementing the add_entry request (todo_web/lib/todo/web.ex)

Listing 11.11. Implementation of add_entry/1 (todo_web/lib/todo/web.ex)

Listing 11.12. Implementing respond/1 (todo_web/lib/todo/web.ex)

Listing 11.13. Retrieving a port from application (todo_env/lib/todo/

web.ex)

Listing 11.14. Using config.exs (todo_env/config/config.exs)

Chapter 12. Building a distributed system

Listing 12.1. Global registration of to-do servers (todo_distributed/lib/todo/server.ex)

Listing 12.2. Locking synchronization in the cache (todo_distributed/lib/todo/cache.ex)

Listing 12.3. Storing data on all nodes (todo_distributed/lib/todo/database.ex)

Chapter 13. Running the system

Listing 13.1. Adding exrm as a dependency (todo_release/mix.exs)

Listing 13.2. Adding runtime_tools as a dependency (todo_runtime/mix.exs)